Insieme
A Compiler Infrastructure for Parallel Programs

PhD thesis in computer science

by

Herbert Jordan

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of philosophy

advisor: Prof. Dr. Thomas Fahringer, Institute of Computer Science

Innsbruck, August 24, 2014

Certificate of Authorship and Originality

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Herbert Jordan, Innsbruck, August 24, 2014

Abstract

Developing programs efficiently utilizing contemporary parallel architectures is complex and time consuming. This is partially due to the inherent problem of revealing parallelism within algorithms and partially due to the fact that programming languages and associated compilation tool have not (yet) been adapted to the fundamental paradigm shift towards parallel systems triggered more than a decade ago.

Available APIs and language extensions for programming parallel architectures are treated by compilers like ordinary libraries utilized by an otherwise sequential host language. Their parallel control flow remains hidden within opaque runtime library calls embedded within a sequential intermediate representation lacking the concepts of parallelism. Consequently, the tuning and coordination of parallelism is clearly beyond the scope of conventional optimizing compilers and hence left to the programmer or the runtime system.

The main objective of the Insieme infrastructure is to provide a platform for researching techniques simplifying the task of developing efficient, scalable and portable parallel programs by gradually off-loading the tuning and coordination efforts to the compiler and the associated runtime system. It is based on a novel, concise, unified, explicitly parallel, high-level intermediate representation making the parallel control flow accessible to the compiler and the associated runtime system. Thus it lays the foundation for the development of reusable, sophisticated, static and dynamic utilities handling the workload and data management, the utilization of heterogeneous hardware and tuning steps for the developer – who, in the end, may only have to focus on revealing a maximum of parallelism.

Within this thesis, the novel design of the Insieme infrastructure is covered. A particular focus is laid on its internal intermediate representation. Additional chapters elaborate analysis and transformation techniques built on top of it. Furthermore, a set of example applications based on the Insieme infrastructure simplifying the development of scalable parallel programs are outlined.

Acknowledgements

When looking back on the path leading to this milestone in my academic and personal life, there are many people without whose support this would have not been possible. Foremost, I'd like to thank my adviser, Prof. Thomas Fahringer, for challenging me into taking up my PhD studies in the first place and for setting up an unique environment granting me the opportunity to develop and pursue a great vision encompassing many of my own ideas and interests over all those years. I'd also like to thank those who lead me to this path, in particular Prof. Aart Middeldorp and his CL department for raising my interest on the theoretical side of computer science as well as my former teacher Mag. Josef Steidl for fostering my interest in computer science almost 15 years ago by teaching me good practices and the importance of a 'clean working style'.

Besides teachers and mentors, fellow students are among the most influential persons in a PhD student's (academic) life. I'm deeply grateful to all the members of the Distributed and Parallel Systems group, in particular to the Insieme team, for providing me with an amazing, fun and fruitful working environment. In particular Peter Thoman, for an extraordinarily efficient and productive cooperation and many interesting, challenging and, if nothing else, entertaining discussions, Simone Pellegrini for his stubbornness forcing me to rethink and attest many of my steps, both of them for teaching me C++, and Luis Ayuso for being a buddy over the last years. Furthermore I'd like to thank Philipp Gschwandtner, Klaus Kofler, Ivan Grasso and the rest of the Insieme team for making work fun as well as for all their relentless efforts contributed to the project.

Finally, I'm in great gratitude to all those people supporting me in my private life throughout those years. My mother Regina for her constant support, my father Herbert for his encouragement and Philipp, Stefan, Lisa and the it-boyz for their friendship. Without you this would have not been possible.

I would also like to thank the external reviewers for providing an additional perspective on this work.

Contents

Chapter 1

Introduction

High-level programming languages have been among the most essential cornerstones for the rapid development of computer technology witnessed over the course of the second half of the last century. The separation of software developers from actual machine architectures by utilizing abstract machines specified by programming languages resulted in portable applications and a vigorous growth in complexity by gradually increased levels of abstraction. On the other end, hardware could be advanced independently of the notation utilized for expressing programs. The enabling transition between high-level languages favored by software developers and the actual underlying hardware architectures has ever since been provided by compilers.

Beside their main role of converting high-level language constructs into low-level sequences of machine processable instructions, compilers have also always been required to conduct optimization steps to produce efficient code. Decades of research and development have been invested in mitigating the *abstraction penalty* introduced by the utilization of idealized, abstract machine models instead of the actual, bare-metal machine specification. A number of contributions regarding techniques for the allocation of registers and the selection and scheduling of instructions have led to tools generating codes exhibiting sufficient quality in terms of computational efficiency for the vast majority of applications.

Nevertheless, over time, computer architectures have undergone significant changes in their quest for achieving everlasting growth in available computational power. Techniques including pipelining, branch predication, superscalar architectures, out-of-order execution, very long instruction words (VLIW), multi-level hierarchies of data and instruction caches, single instruction multiple data instructions (SIMD), simultaneous multithreading (SMT), symmetric multiprocessing (SMP), non-uniform memory access systems (NUMA), and heterogeneous many-core architectures (GPGPUs, FPGAs, big.LITTLE architectures) are employed these days by hardware vendors to provide a maximum of computational power. The proper utilization of such resources for performance-critical programs is the main challenges to be faced by the performance oriented computing community today.

Several of the enumerated techniques are well supported by contemporary compilers. The management of pipelines, branch predication, superscalar architectures, out-of-order execution and VLIW architectures are natural extensions of the responsibilities of an optimizing compiler and can be conducted within the scope of their typical low-level three-address code based intermediate representations (IRs). Consequently, none of those issues require any manual participation of the end user, which, in the context of compilers, corresponds to the software developer. On the other hand, when aiming for performance improving techniques beyond the basic instruction level, e.g. the efficient utilization of data caches and available SIMD instructions, user involvement is still (partially) required these days. While, after almost two decades of research, contemporary compilers are capable of restructuring program code to a certain extent to harness the benefits of cache hierarchies and modern vector units, more complex cases require manual code-restructuring operations and/or user provided annotations to be identified and properly utilized. The improvement of the utilization of even more coarse-grained hardware concepts, including SMT, SMP, NUMA, heterogeneous architectures, and even full clusters of computation notes, is clearly beyond the scope of conventional optimization compilers. The management of applications utilizing such resources is left to the developer. Access is granted by libraries (e.g. Pthread, MPI, OpenCL) or minor language extensions masking the utilization of internal libraries (e.g. OpenMP, Cilk).

Today's hardware is far beyond the available architectures at the time when the foundations for the construction of compilers have been established. In particular, parallel architectures have become the de facto standard for any scale of device throughout the past decade. Yet most programming languages and their associated compilers are still based on sequential application and architecture models. The exposure, management and tuning of parallelism – vital for achieving satisfying resource utilization on all contemporary architectures – is left to the end user although many of the associated tasks are mechanical and could be handled by a future generation of advanced compiler based solutions [72, 82, 74, 97, 78].

The *Insieme Compiler and Runtime* infrastructure presented in this thesis aims for the establishment of a developer platform for tools supporting end users in exposing, managing and tuning parallel programs, thereby utilizing sophisticated compiler and runtime system based techniques. Those are introducing capabilities which are beyond the scope of conventional, pure library based approaches for managing parallel programs. Ultimately, this provides a step toward successfully reducing the *abstraction penalty* for programming complex, parallel, heterogeneous architectures using simple, abstract, high-level programming models. Furthermore, the automated tuning of performance critical aspects can significantly improve the *performance portability* of programs and the productivity of software developers.

1.1 Motivation

The motivation for this present work is best illustrated from a developers point of view. Consider the common case of some simulation frequently encountered within scientific domains. Equipped with a basic idea of the associated algorithm, a software developer is instructed to built a corresponding application. To do so, he picks some sort of high-level language of his choice and starts developing an implementation of the requested algorithm and its associated data structures. Once completed, the resulting program is applied on relevant input data and the job is done – at least one might think so.

Following this standard procedure the resulting application is likely to be a sequential application utilizing only a single core simply due to the fact that most general purpose languages are sequential imperative languages. Thus it only makes use of a small fraction of the available computational power of state-of-the-art desktop and server architectures. Still, to a certain extent, compilers attempt to ensure that this single core is efficiently utilized by applying a variety of low-level optimizations. However, no other resources, in particular no other cores or heterogeneous computing devices within the system, are contributing to the simulation. If the time required for conducting the desired computation is exceeding acceptable boundaries, the focus is quickly drawn onto improving the utilization of all available resources.

Sequential Improvements Let us assume that algorithmic improvements targeting the reduction of the computational complexity of the involved operations have been applied and the resulting program is efficient from an algorithmic point of view. Still, in most cases its sequential performance can be significantly increased by tuning its implementation. For instance, the utilization of the one CPU core that is used at the current state can be further improved by restructuring computationally expensive code regions to access data in a cache-friendly way or to use SIMD instructions. Already these first steps require a fair amount of knowledge regarding the targeted architectures as well as code transformations and their effects. In many cases it also involves the fine-tuning of various transformation parameters (e.g. loop-tiling factors). Furthermore, several of those properties have to be adjusted according to the properties of a specific target system (e.g. its cache geometry). Hence those steps are already influencing the *performance portability* of the resulting application. Yet combined, they have the potential of increasing execution speed by an order of magnitude [49].

Parallelization After algorithmic and sequential code optimizations have been exhausted, the next natural step is to engage additional computation resources like extra CPU cores or GPUs available on the targeted system. Depending on the type of provided resources, the developer has to learn how to use corresponding APIs/language extensions granting access to those by understanding the underlying programming models. He then has to decompose his original implementation such that it can be mapped to the corresponding models, handle communication, synchronization and load balancing issues, and tune parameters exposed by the involved primitives including the number of involved threads, loop scheduling policies, cut-off parameters for nested recursive parallelism, and global and local work group sizes for kernels processed on GPUs. All those steps require a fair amount of knowledge of the underlying concepts and are of course depending on specific properties of the targeted system.

The problem gets even worse when, due to the structure of the targeted system, multiple parallel APIs have to be utilize within the same program. In particular when targeting a distributed memory system consisting of a variety of nodes exhibiting multiple multi-core CPUs and potentially even GPUs, the resulting program code is decomposing the original implementation among a mixture of e.g. MPI, OpenMP, Cilk, Pthread, OpenCL and CUDA calls. Coordinating such a program in a flexible way – such that it can be successfully migrated to another target system exhibiting a different composition of computational resources – is an extremely challenging task easily exceeding the complexity of the actual problem to be solved by the original program. In particular a large portion of the resulting code will be devoted to this task interspersing the program and resulting in hardly maintainable code. Also the reusability and composability of the resulting code fragments is limited due to the required intrinsic management of resources.

Tuning Many of the techniques for improving the performance of applications involve the specification of parameters for the utilized primitives (e.g. tile sizes, number of threads, load balancing policies). Most of those are heavily depending on the context within the application itself, the phase of the application execution, the targeted architecture, and potential external load on the target system. Hence, several of the decisions made regarding those parameters need to be reconsidered at least when moving to a different architecture – ideally continuously in real time during every execution of the program. The latter requires the capability to monitor the program progress as well as its environment to steer its execution – a capability definitely not provided out of the box by conventional general purpose languages nor their associated compiler infrastructures.

API and Tool Support Within contemporary applications all those decisions regarding the management of parallel executions are left to the end user and will therefore be expressed implicitly or explicitly within the application code. Even text books on this subject provide guidelines demonstrating hard-coded decisions regarding the number of threads and the distribution of work load [112]. Consequently, since most parallelism is orchestrated by compiled code within parallel applications, tools influencing those essential decisions are extremely difficult to realize. This is particularly true for codes based on low level APIs such as Pthreads, MPI, OpenCL and CUDA. For the latter two the host-part of an application is essentially the hard-coded dependency graph and scheduling strategy of the involved GPU operations. Altering those requires the program code to be modified – a task only to be conducted by compiler based techniques. Tuning utilities are therefore mostly focusing on properties of the implementations of the utilized APIs, e.g. buffer sizes of message queues or message exchange protocols utilized within MPI implementations [79, 77].

A much more flexible approach is demonstrated by higher-level solutions such as OpenMP. By simply marking a given loop to be executed in parallel, performance critical decisions, including the number of threads or the scheduling policies, are left to the involved sub-systems. This way smart and flexible mechanisms can be employed considering a variety of the static and dynamic influences on those decisions – comfortably concealed from the end user.

Ideally this division of labor between the developer and the compiler could be adapted for other targeted architectures as well. The developer's obligation could be reduced to focus on the identification of potential parallelism and tuning opportunities while the compiler is conducting the necessary mechanical code-restructuring operations and – in cooperation with a corresponding runtime system – the load management and tuning of the involved parameters utilizing sophisticated, reusable techniques and solutions. Such a system would have the potential to enable a broader range of software developers to effectively built applications efficiently utilizing parallel, heterogeneous architectures encountered within all contemporary computational devices. Furthermore, support for new parallel architectures could be integrated by extending compiler and runtime tools – a custom successfully employed for sequential codes and architectures since the dawn of compiler technology – not by manually porting existing applications as it is common practice these days.

1.2 The State of the Art

The ideal development environment outlined within the motivation section comprises compiler and runtime system components capable of steering performance-critical aspects of (parallel) applications over the coarse of their execution. The Insieme research infrastructure, covered in detail within Chapter 2, provides a platform for the development of such tools. One of its central component is INSPIRE – the *INsieme Parallel Intermediate REpresentation* (see Chapter 3) – which unifies the representation of parallel APIs and language extensions, thereby establishing a common foundation for analysis, transformations and program optimizations. The Insieme infrastructure, and in particular INSPIRE and its associated analysis and transformation utilities, is the main topic of this thesis, corroborating the thesis's hypothesis developed at the end of this introduction (see Section 1.4)

To provide a comparison to preexisting systems, the architecture of a few state-of-the-art compiler and parallel library infrastructures utilized in the context of performance oriented and high performance computing (HPC) are outlined within this section.

1.2.1 Compilers

A typical compiler infrastructure consists of five major components. The foundation is laid by an *Intermediate Representation* (IR) and its associated analysis and transformation utilities. Input code is converted by *Frontends* into the IR and *Backends* are utilized to synthesize target code. Advanced abstract operations, e.g. resource management operations, and dynamic interpretation and/or compilation support may be provided by an optional *Runtime System* component. The fifth component, the *Compiler Driver*, is orchestrating the compilation process. In particular, the driver governs the application of compiler passes and hence the sequence of conducted optimization steps. It also constitutes the interface offered to the end user.

The central element of every compiler infrastructure is the design of the intermediate language/representation[1]. It determines the scope of applicable analysis and transformations and hence the potential domains targeted by optimization techniques implemented on top of those. Therefore a special focus will be placed on the IRs utilized by the available infrastructures discussed below. In general, compiler IRs can be associated with one of the following three categories:

- **low-level IRs** - representations that are modeling programs on a level close to machine code. Code is typically represented using three-address code instructions, organized within simple lists or *control flow graphs* (CFGs). IRs on this level provide the necessary foundation for

[1]Strictly speaking, one would have to distinguish the intermediate language (=the formal language and its associated semantic internally utilized to represent programs) and the intermediate representation (=the data structure utilized for its representation). However, since one is the physical manifestation of the other, we use the terms intermediate language and representation interchangeably within this thesis unless explicitly stated otherwise. In particular we use the acronym IR to refer to both of them.

instruction level optimizations, yet are too low-level for a convenient implementation of loop and array level analyses and transformations.

- **high-level IRs** - representations that are close or identical to the structure of the handled high-level source language and implemented based on an abstract syntax tree (AST). High-level IRs are the first choice within source-to-source compilers and provide large high-level optimization potential. Nevertheless, the rich semantic of high-level languages significantly increases the complexity of analyzing this kind of IRs, making complete analysis frameworks based on those de facto unfeasible [114, 65].

- **mixed-level IRs** - this class of IRs is extending low-level IR instructions by a selected set of performance relevant high-level constructs like loops, function calls or array accesses. The additional semantic information can be utilized within analyses and transformations.

Examples for each of those categories will be encountered in the following survey on available compiler infrastructures.

State of the Art Compiler Infrastructures

Among the available compiler infrastructures the GCC and LLVM projects occupy a special position due to their high relevance in software development.

- **GCC** – the *GNU Compiler Collection* – is the default compiler infrastructure for many Linux distributions and supports a large variety of input languages and target architectures. Internally three levels of IRs are utilized – GENERIC, GIMPLE and RTL [61]. GENERIC is an AST like format utilized by a variety of frontends as a common format for constructing full representations of functions before being lowered to GIMPLE, a mix-level IR based on three-address code combined with rudimentary control structures. Depending on the kind of utilized structures high-, low- and SSA-GIMPLE are distinguished. In the GCC backends GIMPLE is further lowered to RTL, the Register Transfer Language. This very low-level IR consists of several hundred instructions organized in roughly a dozen categories.

  Optimizations including dead code elimination, partial redundancy elimination, constant propagation, array based optimizations and vectorization are mostly operating on the target independent GIMPLE level due to the availability of extra high-level control flow information. Machine dependent optimizations focusing on instruction selection and scheduling are applied on the RTL level.

GCC supports the OpenMP language extension for C, C++ and Fortran. The corresponding pragmas are handled by the frontends during the GENERIC to high-GIMPLE conversion by introducing corresponding library calls targeting GOMP library routines [69]. From this stage on the parallel constructs are handled like ordinary function calls by analysis and transformation steps. Also for other parallel libraries no special treatment is conducted.

- **LLVM** is a collection of modular and reusable compiler and tool chain technologies. Unlike GCC the LLVM infrastructure is much closer to a textbook three-phase design (multiple frontends, one IR, multiple backends) by restricting itself to a single, complete intermediate representation. The so called LLVM IR is a low-level assembly like three-address code representation offering a total of 31 RISC like instructions. It is based on an infinite amount of virtual registers and enforces static single assignment form (SSA). A rather surprising trait of the LLVM IR is its strict typing system supporting a variety of primitive types and type constructors as well as the abstraction of calling conventions and the explicit handling of stack and heap memory on the IR level [57]. Nevertheless, loops and other control flow constructs are resolved into corresponding jump instructions as usual for low-level IRs.

  The LLVM project comprises a variety of frontends, backends and optimization passes. Furthermore, the LLVM IR has been designed as a suitable input format for link-time optimizations and lightweight just-in-time compilers.

  As within GCC, parallel libraries are treated like any other library. Hence, their side effects are not considered. Support for OpenMP primitives is currently under development (state March 2014) and will be integrated into the frontend, similar to the GCC approach. Since the LLVM IR is a low-level RISC like representation, constructs for threads and other OS-level objects are clearly beyond its scope.

Beside those two big open source projects a variety of proprietary compilers including Intel's icc [45] and the PGI compiler [39] are available. For those a similar architecture than for the open source projects can be expected. For all of those the utilized internal IR is a rather low-level sequential representation of application code and parallel features are integrated via opaque external library calls.

Nevertheless, within some research compilers, IRs designed for supporting parallel aspects have been realized. For instance, the *Stanford University Intermediate Format* (SUIF) [113, 6] is a mixed-level IR based on conventional low-level RISC-like operations extended by three high-level constructs representing loops, conditional statements and array accesses. This extra information is utilized for array-level optimizations and loop-level parallelization. Furthermore SUIF supports the coverage of multiple translation units within a single instance, thereby expanding the applicability of interprocedural analyses. The SCORE IR [110] is another mixed-level IR connecting higher-level constructs with low-level three-address code operations. However, instead of a conventional control flow graph utilized by other low-/mixed-level IRs the representation is based on a *program dependency graph* explicitly revealing concurrency between code fragments.

All of these approaches remain on the level of three-address code based representations losing structural information present within the original input language [60]. As a result, many other (research) projects relay on the concept of source-to-source compilers for tuning (parallel) input codes.

Source-to-Source Compiler Infrastructures

Unlike conventional compilers accepting high-level code and converting it into a lower-level language, source-to-source compilers are preserving the level of abstraction. In special cases the input language may even be equivalent to the target language. For instance, an implementation of Cilk can be realized by parsing input files and replacing the Cilk keywords by corresponding C-based implementations. The result is pure C code to be compiled by a third-party compiler.

Two of the benefits of source-to-source based compiler solutions are the portability and the support for low-level compiler optimizations inherited for free from third-party compilers utilized for compiling the generated target code. Another is the high-level structure of the internal intermediate representation. The latter is particular useful for the implementation of language features (OpenMP, Cilk) or advanced performance improving compiler transformations commonly expressed based on high-level imperative language constructs including loops, data structures and message send/receive operations [8]. Also, by using a comparable or even the same level of abstraction than the input code, analysis results, performance data or identified problems can be reported back to the end user more naturally.

The development of robust source-to-source program analysis and transformation tools is supported by several infrastructures, including the ROSE compiler infrastructure [88] and Clang, the LLVM C/C++ frontend [3]. Both provide product-quality frontends for their respective input languages (C,C++,Objective-C,Fortran,. . .) producing very detailed, AST-based representations of processed input codes. Those ASTs, which form the IR of derived source-to-source utilities, are designed to model all the details present within the original input code – including comments and code formatting details – to be capable of accurately reproducing the input. They may therefore be utilized as the foundation of refactoring utilities within IDEs. However, as a result, the IR consists of hundreds of different node types and their semantic is based on the corresponding language standards for C/C++/Objective-C, making static analyses challenging, as observed by the developers of Clang [114, 2, 1]:

> *"... Support in the frontend for C++ language features, however, does not automatically translate into support for those features in the static analyzer. Language features need to be specifically modeled in the static analyzer so their semantics can be properly analyzed. Support for analyzing C++ and Objective-C++ files is currently extremely limited, ..."*

One way to reduce the complexity of implementing analyses is to utilize abstractions. For instance, the high-level IR of the Cetus source-to-source compiler [27, 47] introduces a layer of interface based abstractions between their AST like model of translation units and procedures and the implementation of analyses and passes. Thus, more abstract views on the underlying node structure may be utilized whenever applicable. Another approach to facilitate the handling of high-level IRs is to reduce the number of involved constructs, hence node types. The IPR representation [30], an alternative high-level intermediate representation, aims on reducing the number of nodes involved in representing C++ codes within a AST by enforcing regular structures – resulting in ~ 200 node types, compared to the ~ 700 productions of the ISO C++ grammar. The C intermediate language (CIL) [65] goes one step further by eliminating syntactic sugar, semantically overloaded constructs and ambiguities from a conventional C AST resulting in a total of ~ 40 node types. CILpp [114] extends this effort to C++ applications. Clearly, a small, assessable, regular set of node types simplifies the implementation of analyses and transformations. However, none of those representations and their derived static analysis frameworks consider the existence of parallel control flows.

Intermediate Representations for Parallel Codes

Early attempts to add concurrent concepts to SSA based IRs focused on a restricted set of parallel codes based on *cobegin/coend* parallel sections [59, 70]. Pop et al. [84] presented a mechanism enhancing the support for full-blown, OpenMP like, parallel constructs within IRs designed for single-threaded codes. It preserves the optimization potential of classical low-level, sequential optimizations in the presence of parallel constructs. More recently, the Sequential to Parallel Intermediate Representation Extension (SPIRE) has been proposed to augment mid- to high-level IRs to cover parallel constructs [53]. It suggests ten primitives to be introduced into sequential IRs to cover parallel operations. However, SPIRE is lacking the concept of thread groups and hence the possibility of collective work and data sharing constructs. These constructs are required when encoding e.g. nontrivial

OpenMP parallel loops contained within larger parallel regions as well as for structuring nested parallelism. Furthermore, extending conventional IRs by parallel constructs is not resolving the various issues preventing coarse-grained, high-level program restructuring operations, analyses and parallel optimizations as outlined above. Also, none of those techniques are present in available compiler infrastructures.

Finally, on a related area, intermediate representations like the Standard Portable Intermediate Representation (SPIR) [38] have been developed to aid the development of compiler supported accelerator technologies. SPIR provides a standard for representing OpenCL kernels in an input language independent format. The objective is to enable additional languages to be utilized for implementing OpenCL kernels. However, while presenting an IR in the context of parallel applications, SPIR is essentially a variation of the LLVM IR merely modeling the sequential execution of a single OpenCL work item. It does neither (explicitly) cover the parallelism among work items nor the interaction with the host code.

Polyhedral Model based Compilers

A borderline case for an intermediate representation is provided by the *polyhedral model* (PM) [12]. It is based on a mathematical model consisting of a set of linear equations describing the execution of *Static Control Parts* (SCoPs). SCoPs are code regions consisting of arbitrarily nested loops and conditional statements exclusively exhibiting data independent, linear control flow conditions. Hence, the PM can not be utilized as a universal compiler IR. Nevertheless, for code fragments satisfying the SCoP constraints the mathematical formalism provides a powerful foundation for analyses and the composition of transformations [22]. It is therefore increasingly utilized for loop-level optimizations within compiler infrastructures, including GCC [85] and LLVM [37].

Besides its usage as an auxiliary representation for analyses and transformations the polyhedral model is also the foundation for high-level code transformation tools targeting SCoP friendly application domains, e.g. linear algebra or stencil codes. Examples are PLUTO [5] and the CHiLL framework [20]. In particular, within the PM, data dependencies between loop iterations and individual statements can be accurately computed on a per-instance basis. It therefore provides a convenient foundation for automatic parallelization utilities targeting shared and distributed memory systems [26].

1.2.2 Popular Parallel APIs and Language Extensions

Since general purpose languages like C/C++ exhibit no native language constructs for the parallel execution of instructions those features have to be integrated via libraries and/or language extensions. The following enumeration outlines several of the most prominent of those:

- The POSIX Thread library (Pthread) [67] is a standardized interface to the threading and synchronization capabilities offered by modern operating systems. It provides a portable foundation for implementing parallel libraries and runtime systems, yet may also be accessed directly by the end users. However, as a bare-metal interface no support for scheduling or load management operations is provided, except for the OS-level scheduler. Also, since it is based on OS kernel support, its domain is restricted to CPU cores within a single node.

- OpenMP [25] is a mixture of language annotations (pragmas) and library routines providing a higher-level interface to shared memory parallel resources. Concepts including thread groups processing parallel regions, loops, and tasks as well as synchronization primitives like barriers, exclusive access to critical regions, and atomic operations can be utilized at a high level of abstraction, introducing the potential for runtime system and compiler based tuning. Although originally designed exclusively for multi-core shared memory systems, recent extensions introduced in Version 4.0 broadened its scope to accelerators.

- Cilk [17] extends C/C++ by a small set of additional keywords indicating the potential of parallel processable function calls and, in recent versions, loop iterations. In particular within nested recursive applications this simple model provides an elegant mean for expressing parallelism. The Cilk compiler/runtime system influences the execution of the resulting application by determining when to actually utilize the indicated potential parallelism as well as by conducting load balancing among the involved computational resources. Cilk is targeting shared memory architectures.

- The Message Passing Interface (MPI) [92] is a standardized interface for exchanging messages among processes running on different nodes of a distributed memory system. However, it has also been successfully applied to system-on-a-chip architectures providing numerous cores. The offered primitives included low-level point-to-point send/receive operations as well as more complex collective operations for distributing and aggregating data among the involved processes. Due to the end user's direct utilization of the offered communication primitives, the tuning potential of those is, in general, restricted to their implementation. Also, the utilization of local resources within a node can be improved by combining MPI with a shared-memory API. Furthermore, unlike most other APIs, MPI parallelism is always covering the entire application and not restrictable to individual regions. This excludes MPI from being utilized effectively within libraries.

- OpenCL [94] is an API for utilizing heterogeneous computation devices, in particular GPGPUs, in a vendor independent fashion. Similar to Pthreads and MPI it is a low-level API providing the end user a large amount of control. Applications are divided into a host and a kernel part. The host part is a standard application processed by a CPU orchestrating the data movements between devices (in general GPGPUs and CPUs do not share a common address space) and the execution of kernels on those devices. To allow device vendors to flexibly adjust the instruction set of their products, kernels are just-in-time compiled for a particular target device by the host program. CUDA [71] is a proprietary alternative following similar concepts, yet hiding some of the necessary API calls within C/C++ language extensions.

- With the new C++11 language standard libraries supporting higher-level parallel primitives have been introduced. The support comprises platform independent classes representing threads, mutexes, locks and conditional variables – all standardized C++ wrappers of concepts offered by basic system libraries including Pthreads. However, in addition to those, futures and an async operator is offered. This operator triggers a potential concurrent evaluation of an expression similar to the approach taken by Cilk. The compiler and/or the runtime system implementing this operator therefore gains access to similar points of influence as within Cilk applications.

Utilizing parallel libraries for expressing concurrency within applications has several advantages over language or compiler based approaches. For once, new and experimental features can be developed without the requirement of customizing a compiler. Furthermore, library based parallel elements may be flexibly composed to form more abstract parallel constructs. On the other hand the capabilities of libraries are limited in the sense that no C/C++ library managing a user defined task will ever be capable of inspecting the structure of the processed task to derive hints on how to handle it. Nor will a library be capable of rewriting a C/C++ code fragment to be processable on a GPU, as it is partially supported by the new OpenMP 4.0 standard. This kind of operations require the analytic capabilities and influence of a compiler. However, as it has become obvious within the previous section, their IRs – their foundation for reasoning about programs – are in general not aware of parallel control flows at all.

1.3 Open Problems

Within the motivation section several issues to be faced by developers aiming for an efficient utilization of contemporary architectures have been outlined. Ideally compiler infrastructures and associated parallel APIs would provide solutions for those problems. However, as has been described in the state-of-the-art section, their development is (naturally) lagging behind. Among the biggest shortcomings are:

- **Sequential IRs:** Conventional compilers treat parallel APIs and language extensions like ordinary library calls, manifested by invocations of opaque runtime system routines embedded in an purely sequential intermediate representation. Consequently, the tuning and coordination of coarse-grained parallelism is clearly beyond their scope. This obligation is left to the API implementations and/or the end user. Yet library implementations lack the analytic power and influence of compilers, resulting in natural limitations of their capabilities.

- **Unsuitable IR abstraction levels:** Conventional compilers like GCC or LLVM are based on low-level three-address code based representations since those provide the highest flexibility for instruction level optimizations and a convenient, uniform and compact basis for program analysis. However, instruction level IRs are not an adequate format to deal with coarse-grained thread-level parallelism and the associated data management. Available high-level IRs including the ROSE IR or the Clang AST, on the other end, are too close to their input languages and their plethora of language constructs to be effectively statically analyzed. Research approaches like CIL providing streamlined high-level representations of input codes are working toward an adequate level of granularity, yet are still ignoring parallelism.

- **Lack of global perspective:** Most "global" compiler optimizations on low-level IRs are referring to full-procedure optimizations. Interprocedural analyses and optimizations are less frequently encountered. Furthermore, the translation unit (TU) oriented compilation of codes is narrowing the focus of compilers onto individual TUs, yet the control flow within the final application may frequently cross translation unit boundaries. In particular, this applies to parallel control flows implemented utilizing library support. Link-time-optimizations as they are offered by e.g. LLVM have the potential to overcome these limitations. However, at this stage, the input code has already been lowered to the level of instructions. Consequently, coarse-grained thread-level manipulations and the reorganization of data structures is beyond the scope of these approaches.

- **Lack of hardware abstraction:** Many of the available APIs and language extensions are designed for a specific type of hardware architecture, e.g. OpenMP for shared memory, MPI for distributed memory systems and CUDA for accelerators. Consequently the utilized programming models and the set of provided constructs result in applications fitting those targeted systems. However, parallelism within the program code – which would actually be target system independent – is in general not expressed in a generic way such that it may be automatically ported to a different type of system.

- **Hybrid applications:** State-of-the-art parallel APIs and languages have been designed for specific hardware concepts offering parallel processing capabilities (distributed memory systems, shared memory systems, GPUs,. . .). However, contemporary architectures frequently provide a mixture of those elements resulting in the requirement of composing multiple APIs within a single application to harness their full potential. In particular on clusters, hybrid codes utilizing MPI and OpenMP or OpenCL can be encountered [89]. Yet the implementations of the involved libraries are not interacting with each other to perform coordination efforts. This obligation is left to the programmer.

- **The parallel composition problem:** Due to the necessary intrinsic implicit or explicit resource management of parallel codes based on the available APIs, their composability is limited. MPI codes, for instance, depend on *single-program, multiple data* parallelism effectively excluding its utilization within flexibly composable libraries. Also, combining libraries utilizing different parallel APIs (e.g. Cilk vs. OpenMP vs. C++11 threads) results in an uncoordinated state where different parts of the same application may sacrifice efficiency or even disturb each other by competing for resources. Consequently general purpose libraries are rarely parallelized. Parallelization is left to the top-level client code utilizing them. However, in a world where the performance of architectures is increasingly dependent on maximizing the expressed concurrency within applications, this approach may no longer be sufficient.

- **Lack of research infrastructures:** While there are sophisticated infrastructures for researching sequential program optimizations or codes fitting the polyhedral model, parallel optimizations are restricted to runtime systems, library approaches or ad-hoc implementations of source-to-source transformations. A common, unified formalism for representing and reasoning about parallel applications would provide the foundation for reusable, sophisticated analysis and optimization tools and provide a valuable platform for future research and development in the area of parallel languages and associated compiler technologies.

Some of those problems, in particular the software-engineering issues including the lack of hardware abstraction and parallel composition, are targeted by a new generation of parallel languages, including X10 [19], Habanero-Java [10], Habanero-C [28], Charm++ [52] and Chapel [18]. In general those are based on source-to-source compilers translating their input codes to conventional C++ or Java codes referencing required runtime system functionality. The major issue regarding those approaches, however, is the dependency on a new programming language and the associated coding and porting effort for existing libraries and applications. The Insieme project follows a different approach based on C/C++ based mainstream APIs and language extensions.

1.4 Thesis Hypothesis

This thesis is based on the following hypothesis which drove all the associated work:

> *A novel source-to-source compiler architecture based on unified, concise, high-level, holistic and explicitly parallel program models can open up a whole new level of influence of the compiler and the runtime system on the performance of (parallel) programs. This influence can provide the foundation for the development of tools gradually off-loading the load management, tuning and coordination efforts from the software developers to the compiler and the associated runtime system resulting in increased productivity and performance portability.*

The Insieme project corroborates this hypothesis by constituting such an architectures. It aims on the establishment of an infrastructure for the static and dynamic tuning of parallel applications processed by heterogeneous architectures. It is based on a unified, concise, high-level, language and API independent, parallelism-aware compiler IR for static analyses and code transformations as well as a unified, dynamic program model utilized by an associated runtime system for managing the execution of parallel codes within distributed, heterogeneous environments. In this infrastructure, supported parallel APIs are modeled by explicit parallel constructs within the compiler IR reflecting their effects. Parallel APIs are hence no longer mere external libraries. Furthermore, by unifying their representation, compatibility among APIs is achieved. Also, since within the runtime system all parallel operations are managed by a common runtime instance, the coordination among involved APIs is implicitly established. The full system can be considered as a holistic implementation of a variety of state-of-the-art parallel APIs and language extensions realized by utilizing compiler and runtime system based techniques. Furthermore it serves as a research platform providing the foundation for developing advanced parallel language constructs, static or dynamic program analyses, optimizations and application tuning strategies. In particular hybrid optimizations based on compiler aided dynamic runtime optimizations may be realized.

The overall system design of the Insieme infrastructure and in particular the Insieme compiler IR (INSPIRE) and its associated tools is discussed in detail within the following chapters.

1.5 Organization

This thesis is structured into five major chapters. Chapter 2 provides an overview on the Insieme project with a particular focus on the runtime system and its dynamic program model in Section 2.4. Furthermore, the role of the Insieme Compiler and the Insieme Parallel Intermediate Representation (INSPIRE) is covered. A detailed formal specification of the latter is the focus of Chapter 3, encompassing syntactic and semantic aspects as well as examples demonstrating its applicability for accurately modeling prominent constructs encountered within a variety of parallel languages and APIs. It is followed by Chapter 4, elaborating techniques devised for analyzing programs within the Insieme compiler and Chapter 5 introducing utilities developed for transforming INSPIRE codes. Together those utilities provide an extendable tool set offered to developers to built Insieme based utilities for managing and tuning the execution of performance-critical codes. Examples of such utilities are covered in Chapter 6 by discussing a verity of Insieme based research work and tuning utilities established in the context of this thesis. Those demonstrate the suitability of the Insieme architecture, its high-level program models, and its associated tool sets for conducting research related to the development and tuning of scalable, parallel programs.

Chapter 2

Insieme

The content presented in this thesis has been researched and developed to provide the foundation of the Insieme project [29]. Consequently, the requirements on the presented techniques and solutions have been heavily influenced by the project's mission statement and the targeted fields of application. Hence, to establish context, a general introduction on the Insieme project is provided within this chapter. It starts by outlining its mission statement, its architectural orchestration of the involved software entities, and a list of potential applications. It is followed by a high-level introduction to the two main components of the project, the *Insieme Compiler* and the *Insieme Runtime System*.

The development of the Insieme project and the architecture of the involved components are among the major contributions made during the course of the research providing the foundation of this thesis.

2.1 Contributions

The major contributions of this chapter are:

- the development of a novel infrastructure comprising a compiler and a runtime system component for analyzing, transforming and tuning coarse grained parallel programs statically – during their compilation – as well as dynamically – during their execution – utilizing information obtained by compiler based analyses and the observation of the actual program execution as well as the execution environment (Section 2.2.2)

- the establishment of an interface to forward information and tuning options from the static context of the compiler to the dynamic decision making routines of the runtime system to extend the knowledge and the influence of the latter on the structure and performance of a program execution (Section 2.4)

- the specification of a novel, universal runtime system model for heterogeneous, parallel programs targeting architectures comprising shared and distributed memory spaces, general computational resources (e.g. CPU cores) and specialized accelerators (e.g. GPUs); the model provides an abstract view on the actual architecture as well as the processed program and its execution state (Section 2.4.1)

In the context of this thesis, this chapter introduces a novel source-to-source compiler architecture following the criteria of the thesis's hypothesis, which is utilized as an example to corroborate the hypothesis's validity.

2.2 The Insieme Project

The Insieme project has been established at the *University of Innsbruck* to facilitate the research on static and dynamic code optimization and tuning techniques for parallel codes running on homogenous and heterogeneous systems. Since then it has evolved into a sophisticated compiler and runtime system infrastructure serving as the basis for the implementation of a variety of state-of-the-art parallel APIs and language extensions. Those implementations provide the foundation for ongoing research in the area of parallel languages and optimizing compilers.

2.2.1 Mission Statement

The goal of the Insieme project comprises two major elements:

1. the establishment and continuous development of a platform providing a unified implementation of state-of-the-art parallel language extensions and APIs based on compiler and runtime system components facilitating the research of static and dynamic code optimizations for parallel real-world applications

2. the utilization of the established infrastructure for researching and developing tuning techniques, language extensions, APIs and tools improving the performance and the performance portability of parallel applications as well as the productivity of software developers by simplifying the efficient utilization of contemporary architectures and thus the effective development of parallel programs

The first objective implies the establishment of a compiler and runtime system infrastructure based on a unified formalism for representing parallel codes based on a variety of different parallel APIs – in particular including C/C++ based OpenMP, Cilk, OpenCL and MPI. This unified formalism provides the foundation for the interoperability of the supported parallel APIs. Furthermore, it serves as the basis for a common, API independent

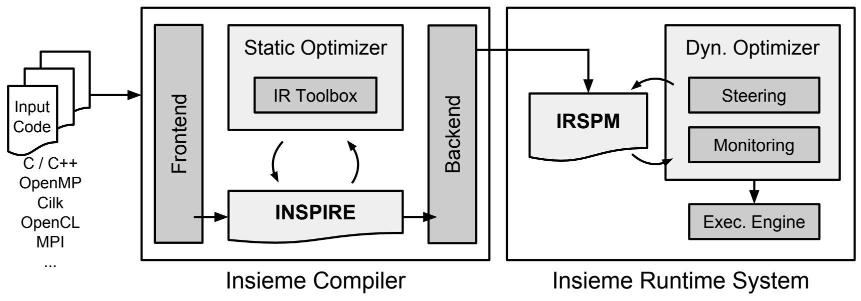


Figure 2.1: Overview of the Insieme architecture.

set of static and dynamic analysis and manipulation utilities. To facilitate research on top of the Insieme infrastructure, interfaces granting access to static program analysis and transformations as well as on dynamic scheduling decisions, load balancing policies, application steering facilities and real-time monitoring data are provided. Furthermore, support for *compiler-aided dynamic application tuning* is provided. In those scenarios compiler components utilize their analytical capabilities and influence on the processed application to provide additional information and/or tuning opportunities to the dynamic decision making processes of the runtime system which therefore can conduct more informed decisions.

Based on the established compiler and runtime system infrastructure which exposes interfaces to influence the compilation and execution of parallel applications, optimization strategies following a variety of approaches are researched. In the context of the Insieme project analytical, search based and machine learning based static and dynamic optimization strategies have been successfully applied to a variety of tuning problems [97, 49, 99, 36, 78]. Furthermore, for some applications, the conventional focus of reducing the overall execution time of an application has been extended to consider additional objectives, including the parallel efficiency or the energy consumption of applications, by formalizing the overall tuning problem as a *multi-objective* optimization problem. Finally, to provide end users access to those advanced tuning capabilities, corresponding languages and API extensions are being investigated.

2.2.2 Architecture

The foundation for the features and capabilities of the Insieme infrastructure is laid by its architecture. An overview on the essential components is illustrated in Figure 2.1. The two main components are the *Insieme Compiler* (the *compiler*) and the *Insieme Runtime System* (the *runtime*). Both components are based on a unified formal representation of applications. Within the compiler the *INSieme Parallel Intermediate REpresentation* (INSPIRE) is utilized while the runtime is based on the *Insieme Runtime System Pro-*

gram Model (IRSPM). Both provide the foundation for reasoning and handling parallel applications within their respective contexts. Consequently, INSPIRE, the static application model within the Insieme infrastructure, is a unified, language and API independent, high-level, full program[1] representation focused on supporting analyses and code transformations within the compiler while IRSPM, the dynamic application model, is a evolving collection of annotated *work* and *data items* describing the state and tuning opportunities of a parallel application while being processed by the runtime.

A typical work flow through the compiler proceeds as follows: A given input code is passed to the compiler which will utilize its frontend to convert it into the INSPIRE format. In the course of the conversion, parallel constructs of supported APIs are translated into explicit parallel constructs provided by INSPIRE. Currently C/C++ based OpenMP, Cilk, OpenCL and MPI are (partially) supported. Once converted into the internal IR, a series of analysis and transformation passes offered by the *IR toolbox* may be applied on the processed program. The actual operations are determined by a research or tool specific static optimizer component. After completion, the program is forwarded to the backend which decomposes the program into *work* and *data items* according to the IRSPM and synthesizes C/C++ code utilizing constructs offered by the runtime system to describe the decomposed application. The obtained C/C++ *target code* is then compiled to an executable binary utilizing some third-party compiler (not shown within Figure 2.1). When running the resulting application on a target system, the runtime system is in control of the execution – not the actual application. The runtime – in particular the customizable dynamic optimizer within it – decides when and on what available resource which work item of the decomposed application will be processed by utilizing the application steering facilities provided by the IRSPM. The runtime also manages the location of data shared between work items in the form of data items, is aware of the dependencies between work and data items and may decide to distribute workload and data among distributed memory systems and/or accelerators. To close the control loop required for effective application tuning, monitoring facilities covering the processed application itself as well as the state of the system environment (e.g. external load) are provided to the dynamic optimizer.

When utilizing the Insieme infrastructure for researching optimization techniques or building tuning utilities, developers may freely adjust the static and dynamic optimizer components within the compiler and the runtime system. An example setup is shown in Figure 2.2. The illustrated setup has been utilized for tuning parallel codes considering multiple objectives [49]. The left side sketches the operation of the static optimizer within the compiler while the right side covers the internals of the dynamic opti-

[1]opposed to conventional translation unit focused representations

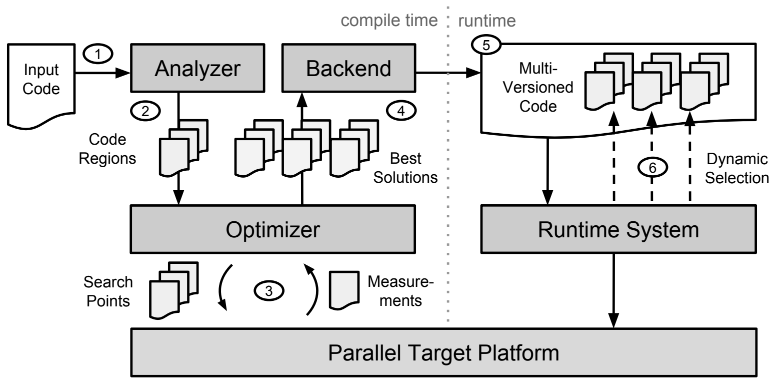


Figure 2.2: Example utilization of the Insieme infrastructure.

mizer. After loading input codes (1) the static optimizer identifies tunable code regions utilizing analysis form the IR toolbox (2) and forwarding it to a research specific optimizer. In this particular case the optimizer is a search based optimizer depending on the evaluation of individual versions of the identified code regions. Therefore sets of code versions (search points) are iteratively tested by applying the corresponding code transformations on the input code, creating binaries and running them on the target system to obtain performance data (3). Those steps are conducted utilizing the compiler backend and the monitoring infrastructure of the runtime system. Since in this scenario multiple tuning objectives have been considered, the optimizer yields a set of Pareto efficient solutions instead of a single solution obtained in a conventional case (4). Those, together with the input program are forwarded to a code synthesizer creating a target code exhibiting multiple implementations of the tuned regions – one for each element within the solution set (5). Each of the versions is annotated with information regarding the specific trade-off of the objectives it is manifesting. Finally, during execution, the dynamic optimizer within the runtime may flexibly select the version of the tuned code region fitting the current system state best (6). The decisions is based on information regarding the system state obtained via the available monitoring facilities. More details on this specific use case of the Insieme infrastructure are covered in Section 6.2.

2.2.3 Applications

The Insieme infrastructure can be utilized for researching a variety of aspects associated to the development and tuning of parallel applications. In particular those include:

- **Conventional high-level compiler optimizations:** By focusing on the compiler component, the high-level compiler IR, its explicit representation of parallel constructs and its full-program scope can be utilized for researching advanced static analysis, code transformations and optimization strategies.

- **Conventional runtime system tuning:** Similarly, focusing on runtime only aspects, the provided control over parallel applications enables research on scheduling, load balancing and resource management techniques for parallel applications – independently of the actual APIs utilized within input codes.

- **Compiler-aided dynamic application tuning:** The full power of the Insieme infrastructure can be harnessed when combining the analytic power and code manipulation capabilities of the compiler with the knowledge regarding the dynamic state of the application and the target system visible to the runtime system in order to conduct a mixture of static and dynamic application tuning operations. In those approaches the compiler component prepares and annotates tuning options to be dynamically utilized upon demand by the runtime system. An example configuration following this approach has been outlined within the previous architecture section.

- **Parallel language research:** Furthermore, the Insieme infrastructure can serve as a platform for researching new parallel APIs and language extensions simplifying the implementation and tuning of parallel applications. For instance, new OpenMP constructs may be supported by extending the corresponding frontend to conduct a proper encoding of their effects into INSPIRE code fragments. Due to the unified internal representation the rest of the system, including analysis, transformations, the backend and the runtime system, will be able to process those extensions. Since INSPIRE is language independent one might even implement frontends for *domain specific languages* (DSLs) exhibiting parallel features. Once encoded utilizing INSPIRE, the rest of the system's capabilities are available for those DSLs.

- **Compiler related technologies:** Finally, the novel nature of the explicitly parallel IR introduces new challenges in the area of static program analysis that are not met by conventional control-flow graph based program analysis. Advanced solutions for handling those accurately may be researched on top of real-world applications based on the available infrastructure. The results could be utilized e.g. for static, API independent dead lock or race condition detection utilities.

Only a few of the mentioned research directions have been explored so far in the context of the Insieme project. Some of those activities and the resulting techniques are covered in Chapters 4 - 6.

2.3 The Insieme Compiler

In the previous section a brief overview on the Insieme architecture has been provided. In this and the following section more details regarding the two main components – the compiler and the runtime – are presented.

The Insieme compiler is a modular toolkit of compiler related components supporting the creation of source-to-source compiler technology based utilities. As has been stated above, the central element of the compiler is INSPIRE which is covered in great detail in Chapter 3. Based on this unified representation a variety of utilities organized within several modules have been developed. Together they are constituting the Insieme compiler infrastructure. The modules include:

- **The Frontend:** The standard Insieme compiler frontend is based on Clang [3]. C/C++ input codes are parsed and checked for syntactic errors before being converted into INSPIRE. The frontend may thereby convert a large number of translation units, before merging them into a full IR program. After this initial conversion, individual API specific passes may be applied on the resulting program to replace opaque API calls with corresponding explicit IR constructs. At its current development state (March 2014) the available passes encompass (partial) support for OpenMP, Cilk, OpenCL and MPI. The result is a complete, high-level, unified representation of a full parallel program exhibiting explicit parallel control flow constructs. As a module the utilization of the frontend is optional and may hence be substituted by an alternative source of an IR program, e.g. a parser for DSLs.

- **The Core:** The core essentially encompasses the data structures and associated utilities of the INSPIRE implementation. Beside the IR node data structures and definitions of IR primitives and extensions (see Chapter 3) essential inspection and transformation utilities are included. In addition, checks verifying the validity of IR code fragments, in particular covering type checks and the scope of variables, are included. These checks have proven to be vital aids when developing IR based analysis and transformations. Also, utilities simplifying the construction of IR language fragments, including an IR parser providing the convenient option of constructing complex IR fragments utilizing a simple, intuitive string representation, are offered. Finally, load/store utilities for saving IR codes in files are included as well.

- **The Backend:** The backend is a modular kit for assembling code generating components. The current implementation covers three configurations – one generating code suitable for the Insieme runtime system, another for embedded OpenCL kernels and a third for stand-alone sequential applications without any dependencies to runtime components. The latter is eliminating any parallelism from an application before synthesizing pure C/C++ code. It is mainly utilized for development purposes, the generation of sequential reference codes and as a foundation for some compiler driven dynamic program analysis.

- **The Analyses:** This module comprises utilities for advanced IR based static and dynamic code analyses. The support ranges from simple static code feature extractors over a conventional data flow analysis framework (DFA), a comprehensive constraint based analysis framework covering data and (parallel) control flows (CBA) to polyhedral model based analyses (PM) and dynamic analyses monitoring actual program executions. Details are covered in Chapter 4.
- **The Transformations:** IR based transformations may be implemented by manipulating the IR directly utilizing primitives offered by the compiler core or by more abstract means covered by the transformation module. To that end, a system for describing code manipulations based on patterns similar to regular expressions is offered as well as an infrastructure for describing code manipulations based on the polyhedral model. Finally, a framework for combining parametrized transformations into code transformation scripts is included. Details are covered in Chapter 5.
- **The Driver Toolkit:** Finally, a collection of utilities simplifying the development of tool or research specific compiler drivers is offered by the driver module. In particular utilities for parsing command line parameters as well as implementations of various optimization strategies contributed by research activities conducted on top of the Insieme infrastructure are available.

Fundamentally, the Insieme compiler is – unlike GCC or other state-of-the-art compilers – not an application that can be controlled by passing command-line parameters. Instead it is a collection of programming libraries and utilities providing a simple infrastructure for building these kind of tools. Nevertheless, the driver module includes the *insiemecc* driver which can be utilized as a drop-in replacement for gcc/g++ within e.g. make-file based build environments to compile parallel codes to be managed by the Insieme runtime system.

2.4 The Insieme Runtime System

Similarly to the compiler, the Insieme runtime system is based on a unified model for parallel programs. However, unlike the compiler IR, the runtime model is designed to facilitate the management of the parallel execution of compiled program code fragments. In this section a formal description of the corresponding models and a brief summary of the implementation of the runtime system is provided.

Note: In this section, only a general overview on the the Insieme runtime system and its associated components is covered. The focus is placed on the involved formal models. More details regarding their implementation and related research results can be found in Peter Thoman's thesis covering the Insieme runtime system [80].

2.4.1 The Program Model

The Insieme Runtime System Program Model (IRSPM) is the representation utilized by the runtime system to manage program executions. It is centered around two major constructs:

- **data items:** blocks of structured data distributed throughout the system
- **work items:** chunks of parallel work units performing operations on data items

Every application starts with a single work item encapsulating the processing of the entry point of the original application (e.g. the main function). This work item is assigned to one of the available hardware threads and started to be processed. Over the course of its execution it may create, access, update and destroy data items as well as spawn new work items conducting concurrent operations on the shared set of data items. However, the runtime system manages the distribution of data items and work items among the available resources. For this purpose, information regarding the relation between work and data items in addition to data on the available hardware is required – and exposed by the following model covering those entities at a suitable level of abstraction.

Data Items

In a first step we will establish a definition for data items.

Definition 2.1 (data items)**.** A *data item* d is a n-dimensional array of homogeneous elements and identified by the 3-tuple

$$d = [id, size, \tau] \in \mathbb{N} \times \mathbb{N}^* \times \mathbb{T} = D$$

where $id \in \mathbb{N}$ is its id, $|size| \in \mathbb{N}$ is its number of dimensions, $size \in \mathbb{N}^*$ is a tuple defining the size of each dimension and $\tau \in \mathbb{T}$ defines the type of the elements stored within the data item where $\mathbb{T}$ is the set of all types. The set of all data items is denoted by D.

Example 2.1 (data item)**.** For instance, a data item $d_1 = [4, [2, 3], int]$ describes a 2-dimensional 2-by-3 array of integers with the id 4 while a data item $d_2 = [7, [], double]$ references a data item containing a scalar (0-dimensional) double value with the id 7. The element type may also be a composed type. For instance, the data item $[5, [3], \text{struct p \{ int x; int y; \}}]$ describes a 1-dimensional array of 3 pairs of integers.

The data item model fits the dominant data structure utilized by parallel programs for distributing data – arrays. More complex data structures, including lists, trees, DAGs, graphs or sets have to be modeled utilizing sets of data items based on this model if they are required to be exchanged between work items. Since in C/C++ every data structure is eventually reduced to scalars, struts or arrays, this does not impose any limitation. However, future work on the runtime model may focus on the integration of improved support for additional kinds of data structures to reduce management overhead as required.

Data items are utilized for addressing structured blocks of data within the system. However, the actual value stored within those is modeled by *value assignment functions*. Such a function maps data items to elements of their respective *value domains*, which have to be defined first.

Definition 2.2 (value domain)**.** Let $dom(\tau)$ be the domain of a type $\tau \in \mathbb{T}$. Further, let $dom(\tau, size)$ defined by

$$dom(\tau, size) = \begin{cases} dom(\tau) & \text{if } size = [] \\ dom(\tau, [s_2, \ldots, s_n])^{s_1} & \text{if } size = [s_1, \ldots, s_n] \end{cases}$$

be its extension for fixed-sized nested arrays. The set of all values is denoted by $\mathcal{V}$ and defined by

$$\mathcal{V} = \bigcup_{\tau \in \mathbb{T}} \bigcup_{size \in \mathbb{N}^*} dom(\tau, size)$$

which corresponds to the union of all fixed-sized, nested arrays over any potential type $\tau \in \mathbb{T}$.

Based on value domains, value assignment functions are defined as covered in the following definition.

Definition 2.3 (value assignment functions)**.** A *value assignment function* $\nu : D \to \mathcal{V}$ is a partial function mapping data items to values such that

$$\forall d = [id, size, \tau] \in \nu^{-1}(\mathcal{V}) \ . \ \nu(d) \in dom(\tau, size)$$

where $\nu^{-1}(\mathcal{V})$ denotes the preimage of the function ν. Hence, every data item d in $\nu^{-1}(\mathcal{V})$ is mapped by ν to a value of its corresponding $|size|$-dimensional domain.

Example 2.2 (value domains)**.** As an example, the domain of integer values $dom(int)$ may be defined by the range $[-2^{31}, \ldots, 2^{31} - 1]$. The domain of 1-dimensional integer arrays of size [4] is defined by

$$dom(int, [4]) = dom(int)^4 = [-2^{31}, \ldots, 2^{31} - 1]^4$$

and corresponds to all sequences of integers of length 4. Similar, the domain of 2-dimensional arrays of size $[10, 15]$ is defined by

$$dom(int, [10, 15]) = dom(int, [15])^{10} = ([-2^{31}, \ldots, 2^{31} - 1]^{10})^{15}$$

corresponding to all sequences of length 15 consisting of sequences of integers of length 10. The domain of integer scalars, hence $size = []$, is given by $dom(int, []) = dom(int)$ as expected.

Example 2.3 (value assignment function)**.** The partial function $\nu_1 : D \to \mathcal{V}$ mapping $[7, [2], int] \mapsto [1, 2]$ is a valid value assignment function while $\nu_2 : D \to \mathcal{V}$ mapping $[7, [2], int] \mapsto [2]$ is not since the dimension of the assigned value does not match the size of the data item.

Unlike the data item model, which can be directly utilized for an actual implementation, the *value assignment function* is merely an auxiliary construct utilized by this formal description of the IRSPM. In the actual implementation realized by the runtime system data items include C-pointers to memory blocks containing the actual values. However, data items are not necessarily stored within continuous blocks of memory. Sub-structures may be distributed throughout the system to utilize e.g. distributed memory systems, device memory on GPUs or to harness the performance benefits of NUMA systems. The fragmentation of data items is thereby controlled by its regular n-dimensional grid structure and explicit data requirements stated by work items operating on those.

Definition 2.4 (data requirements)**.** A *data requirement* is a triple

$$[d, r, a] \in D \times \mathbb{N}^{2^*} \times \{RO, RW, WO\}$$

where $d = [id, size, \tau] \in D$ references a data item, $r \in \mathbb{N}^{2|size|} \subset \mathbb{N}^{2^*}$ a $|size|$-dimensional sub-range of the full data item and a the requested access rights (read-only, read/write, write-only).

Example 2.4 (data requirements)**.** For instance, a data requirement

$$[[7, [10, 12, 14], int], [[0, 5], [2, 8], [4, 6]], RO]$$

describes the requirement of read-only access to the block

$$[0, \ldots, 5] \times [2, \ldots, 8] \times [4, \ldots, 6]$$

of the 3-dimensional data item with the id 7 referencing a $10 \times 12 \times 14$ array of integer values.

If, for instance, a work item requires read-only access to a sub-section of the data stored within a data item, it can specify the corresponding dependency. The runtime system could then look up the current location of the data item and if possible co-locate the work item and the requested fraction of data on affiliated resources – e.g. on the corresponding node within a distributed memory system. The necessary coordination of the state of code fragments is managed similar to the MESI protocol [75, 43] for realizing cache coherency – yet on a much more coarse grained level of array-sub-sections. To ensure the absence of race conditions on a fine grained, element wise level as well as the proper definition of requirements is the obligation of the processed program code.

Work Items

Work items are the active part of the program model conducting operations on data items. Each work item may thereby exhibit multiple, semantically equivalent implementations customized for a specific kind of objective. For instance, a work item may have three implementations assigned, one for being sequentially processed on a single CPU core, another for a parallel processing on multiple CPU cores and a third for running the same computation on a GPU. Additionally, as has been stated in the previous section, every work item has to provide means for obtaining data dependencies as they are required for the data item management. The necessary formalism to incorporate all these requirements is covered in the following definitions.

Definition 2.5 (work item execution state)**.** The set

$$W_S = \{\text{init}, \text{ready}, \text{running}, \text{suspended}, \text{resumable}, \text{done}\}$$

denotes the set of *work item execution states.*

Definition 2.6 (work item)**.** A *work item (instance)* is a 5-tuple

$$w_i = [\text{id}, \text{desc}, \text{state}, \text{range}, \text{param}] \in \mathbb{N} \times W_D \times W_S \times \mathbb{N}^2 \times D$$

where id $\in \mathbb{N}$ is its id, desc $\in W_D$ the work item description summarizing its implementation (see Definition 2.7), state $\in W_S$ its execution state, range $\in \mathbb{N}^2$ the range of parallel operations covered by the work item and param $\in D$ a single data item attached as a parameter to the work item instance.

Definition 2.7 (work item description)**.** Let $\mathcal{M}$ denote an arbitrary set of *meta-information.* For a work item $w_i = [i, d, s, r, p]$ the *work item description* $d \in W_D$ describes the implementation of the work item w_i and is given by a triple

$$d = [r_d, \text{impls}, m_w] \in R_D \times 2^{I \times R_H} \times \mathcal{M}$$

where r_d is a *data requirement function* of the set

$$R_D = (\mathbb{N}^2 \times \mathcal{V}) \to 2^{D \times \mathbb{N}^{2^*} \times \{RO, RW, WO\}}$$

mapping the range $r \in \mathbb{N}^2$ of the work item w_i and the value $\nu(p) \in \mathcal{V}$ of its parameter $p \in D$ to a set of data requirements $r_d(r, \nu(p))$. The set of implementations

$$\text{impls} \in 2^{I \times 2^{\mathcal{H}} \times \mathcal{M}}$$

consists of triples $[i, r_h, m_i] \in I \times 2^{\mathcal{H}} \times \mathcal{M}$ where i is an *implementation* of a procedure accepting the work item range r $\in \mathbb{N}^2$ and the value $\nu(p) \in \mathcal{V}$ of the work item parameter $p \in D$ for processing the operations represented by the work item instance w_i and the *hardware requirements* $r_h \subseteq \mathcal{H}$ summarize the resource requirements demanded by the associated implementation where $\mathcal{H}$ is an abstract set of system resources. Finally, the meta-information elements $m_w \in \mathcal{M}$ and $m_i \in \mathcal{M}$ contain additional, generic information associated to the enclosing work item description or implementations respectively.

Essentially, while work items are dynamic elements representing units of work, work item descriptions are summarizing their parametrized static properties, including their implementations and dependencies – much like the relation between objects and classes in object oriented programming languages.

The integration of the generic meta-information has been covered for completeness, yet it does not effect the runtime system's program model. Meta-information is utilized for forwarding statically obtained information regarding work items and their implementations from the compiler to the runtime and may be extended and/or customized for specific use cases. Examples include details regarding the tuning trade-offs embodied by specific work item implementations, data enabling the estimation of computational costs or details regarding the nested parallel structure of the associated work item. Those particular examples and their utilization are covered in detail in Chapter 6.

Example 2.5 (work items)**.** Let i_1 and i_2 be two semantically equivalent implementations of the same procedure (work item) where i_1 is running on a CPU while i_2 is utilizing a CPU and GPU for its operations. Let the corresponding *hardware requirements* be denoted by

$$r_{h1} = \{CPU\} \subset \mathcal{H}$$

and

$$r_{h2} = \{CPU, GPU\} \subset \mathcal{H}$$

Further, let $m, m_1, m_2 \in \mathcal{M}$ be three arbitrary meta-information entities, $d = [7, [10], int]$ be a data item referencing an 1-dimensional array of 10 integer values and $p = [4, [], \mathit{DataItem}]$ be a data item addressing a scalar storing a reference to the data item d (hence $\nu(p) = d$). A work item with id 5 instantiated by argument p based on the two implementations $i_1, i_2 \in I$ consisting of 10 concurrent operations ready to be processed is represented by

$$w = [5, [r_d, \{[i_1, r_{h1}, m_1], [i_2, r_{h2}, m_2]\}, m], \mathit{ready}, [0, 9], p]$$

where $r_d \in R_D$ is the *data requirement function*. In case every operation j of the covered range of concurrent operations $0 \ldots 9$ depends on read/write access on the j-th element of the data item referenced by parameter p and read access to the full range, the function $r_d : \mathbb{N}^2 \times \mathcal{V}$ is defined by

$$\begin{aligned} r_d((l, u), d) &= r_d((l, u), [id_d, [s_d], \tau_d]) \\ &= \{[d, [[l, u]], \mathit{RW}], [d, [[0, s_d]], \mathit{RO}]\} \end{aligned}$$

Upon processing, the runtime system inspects the work item w, computes its data dependencies, checks the location of the requested data, picks one of the available implementations $i_x \in \{i_1, i_2\}$, e.g. depending on the requested and available resources, and assigns the work item to a corresponding computational unit by running the procedure call $i_x([0, 9], \nu(p))$ on it. Alternatively, the work item w might, for instance, be split into two work items

$$w_1 = [4, [r_d, \{[i_1, r_{h1}], [i_2, r_{h2}]\}], \mathit{ready}, [0, 3], p]$$

and

$$w_2 = [4, [r_d, \{[i_1, r_{h1}], [i_2, r_{h2}]\}], \mathit{ready}, [4, 9], p]$$

by partitioning the work item range $[0 \ldots 9]$ into $[0 \ldots 3]$ and $[4 \ldots 9]$ and assigned the pieces to distinct computational units if multiple resources are available. Note that the resource requirements of the resulting work items are, by design, implicitly reduced accordingly. The fragments w_1 and w_2 may even be processed by selecting distinct implementation, resulting in the partial processing of the original work item w on a CPU and GPU.

Beside the resource requirements, annotated meta-information can be utilized by the decision making processes within the runtime system for gaining extra information on the available options.

The Program Model

A program processed by the Insieme runtime system is constituted by a set of work and data items according to the definitions provided above. Due to this simple global structure multiple applications may even be managed by a single runtime system instance simultaneously since no application specific considerations have to be made. This feature could, for instance, be used for increasing the system utilization if multiple, simultaneously executed parallel programs exhibiting phases of varying concurrency are managed by a single runtime instance [101].

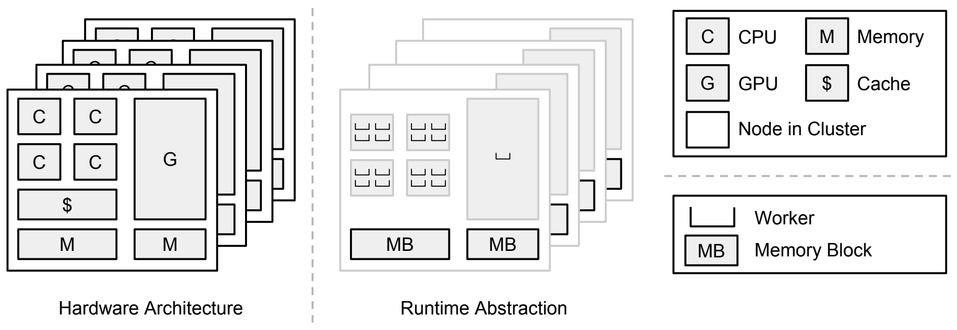


Figure 2.3: Abstraction of resources in the Insieme Runtime System.

2.4.2 The System Model

Besides the program model an abstraction of the available computational resources is required to standardize the interface offered to the dynamic optimizer. The corresponding abstract concepts are

- **Worker:** entities capable of processing work items
- **Memory blocks:** entities capable of storing (fractions) of data items

as illustrated in Figure 2.3. Essentially, for each computational unit (CPU core/hardware thread, GPU, . . .) in the system, a *worker thread* is created. Each of those has an associated queue of work items which are consecutively processed. Newly spawn work items are first enqueued locally but may get stolen by other workers under certain conditions related to their execution state. For instance, work items in their initial state may get stolen to remote worker running on another node of a distributed memory system while suspended work items may only be moved between workers of a single runtime instance due to potential local dependencies (e.g. work-item local data allocated on the heap).

Memory blocks, on the other hand, are the data-equivalent of workers. A block is created for each address space within the system exhibiting distinct access characteristics. Hence, the various levels of device memory on GPUs may form different memory blocks, as well as the different nodes of a NUMA architecture or the various address spaces of a distributed memory system.

While the worker and work item infrastructure is fully implemented in the current development state of the Insieme runtime system, the data items, memory blocks and data requirement functions have so far only been demonstrated within a prototype implementation. Their integration into the Insieme infrastructure is still pending (state March 2014).

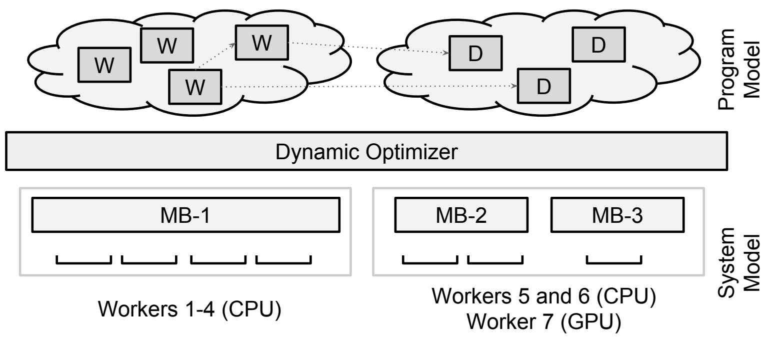


Figure 2.4: Abstract environment as seen by the dynamic optimizer.

Dynamic Optimizers

The role of a dynamic optimizer is illustrated in Figure 2.4 based on the entities defined by the program and system models. Its obligation is to utilize the information provided by a program to map its work and data items dynamically, within the constraints imposed by associated data requirement functions, to the available resources abstracted by workers and memory blocks in order to achieve desired objectives – e.g. the minimization of execution time, energy consumption and/or power dissipation. This component is research specific and may utilize all the work and data item capabilities as required.

2.4.3 Runtime System Components

The implementation of the runtime system comprises a variety of utilities assisting the work of the dynamic optimizer. To complete the coverage of the runtime system a few of those shall be mentioned:

- *Model Entities:* The core functionality of the runtime is formed by the implementation of the entities covered by the program and system models, including work and data items, workers and memory blocks. APIs for interacting with those are offered.
- *Event System:* The runtime system is equipped with a generic event propagation system enabling components to subscribe for events according to the general observer pattern. This infrastructure enables operations to be implemented based on push notifications instead of potentially wasteful pull operations polling for arbitrary events. It also abstracts from required inter-process communications when being utilized within a distributed memory system setup.
- *Monitoring System:* To effectively control non-functional aspects of an application, e.g. its execution time or energy consumption, means for

measuring those metrics are required and provided by the monitoring sub-system. Additionally, hardware and event counters characterizing the execution of applications are offered.

- *Hardware Model:* Finally, for many decisions to be made by the dynamic optimizer, information regarding the infrastructure the application is processed on is required. For instance, information describing the connection between CPU cores and memory blocks or the organization of cache hierarchies may be utilized for distributing data and work items. The corresponding information is provided by this module of the runtime system.

In addition to supporting the implementation of dynamic optimizers, several of the provided utilities may also be utilized for implementing customized higher-level synchronization and communication operations for specific use cases. These higher-level constructs may then be utilized by the backend within synthesized code to improve the performance of those operations.

2.5 Summary

In this chapter the novel architecture of the Insieme system has been presented. Unlike conventional compilers including GCC or LLVM, the Insieme compiler retains a high-level intermediate representation of the processed programs explicitly exposing parallelism – a novel design in the area of general purpose compilers processing parallel program codes [48]. The awareness regarding high-level structures, in particular parallel constructs, opens up a whole new level of influence on the tuning of (parallel) program codes during its compilation and – in combination with the runtime system model – during its execution. The utilization of a sophisticated, unified, program and system model in the runtime system provides the foundation for program steering and tuning opportunities to be exploited by an adaptable dynamic program optimization component. Dynamic optimizer implementations may additionally be aided by compiler derived analysis data to conduct more informed decisions. This novel, close integration of compiler and runtime system components establishes the foundation for innovative, automated, and compiler-aided dynamic program tuning solutions (see Chapter 6).

Chapter 3

The Insieme Parallel Intermediate Representation

As has been outlined by the previous section, the foundation for the Insieme compiler component is laid by its unified, high-level compiler IR [48]. The design of this formal language is covered within this chapter.

3.1 Contributions

The major contributions of this chapter are:

- the formal specification of a novel, unified, high-level compiler intermediate representation that is concise in the number of involved constructs, language, API and programming model independent, and explicitly in the representation of coarse grained, thread or process level parallelism; the specification covers the syntax and semantic of all the involved constructs (Sections 3.4 to 3.7)

- the integration of a flexible extension mechanism into the basic intermediate representation design and its semantic interpretation based on abstract data types (ADTs) as well as its utilization for modeling a variety of common language and API constructs (Section 3.8)

- the demonstration if the intermediate representation's capability of encoding language constructs and primitives offered by (parallel) programming languages, parallel language extensions and (parallel) APIs (Section 3.9 and Section 3.11)

In the context of this thesis and its hypothesis, this chapter provides a detailed example of a compiler IR satisfying the hypothesis's criteria, thereby demonstrating that such an IR can be designed and implemented in the context of a general purpose compiler processing real world parallel programs.

3.2 Design Goals

The basic requirements on the Insieme IR design have been summarized within the previous chapter. It has to provide the foundation for an infrastructure capable of effectively tuning heterogeneous parallel application code. Furthermore, general requirements for compiler IRs have to be considered. Every suitable compiler IR should be

- *Expressive* – the IR must be capable of expressing all relevant features of the input language as well as all aspects the optimizing compiler is intended to tune within the processed codes
- *Analyzable* – the IR should be structured in a way such that analyzes required for the optimization process can be (efficiently) conducted
- *Transformable* – the IR design has to reflect the requirements of an optimizing compiler to efficiently manipulate the IR as well as to convert input codes into the IR and the IR into target code
- *Extensible* – since compiler projects are running for several years, requirements on their IRs are likely to evolve over time. An IR should be adaptable to new requirements with a limited impact on the IR's implementation and related utilities

Several of those criteria are conflicting with each other. For instance, a very expressive, detailed IR like a complete C/C++ AST is harder to analyze than a more restricted IR [65]. Also, realizing an extensible IR infrastructure capable of dealing with evolving requirements requires analysis and transformation utilities to be more flexible and generic than for an environment not offering corresponding facilities. To resolve some of the conflicts, we derived the following principles for the design of INSPIRE:

- *Complete* – any input program shall be representable by a comprehensive, self-contained IR structure covering all the information required to reproduce semantically equivalent target code; in particular, the IR shall not be restricted to a subset of the input codes nor depend on references to external data not covered by the IR
- *Explicit* – all important concepts are covered explicitly at an adjustable level of abstraction to simplify analysis and allow transformations to affect those aspects; e.g. explicit parallelism and memory management
- *Unified* – constructs constituting the same meaning shall be expressed using identical means to foster the re-usability of analyzes and transformations; e.g. barriers or reductions originating from different input languages shall be represented using the same primitives

- *Simple* – IR constructs shall have a precise, non-overloaded interpretation to avoid dealing with special cases; e.g. *for*-loops should be restricted to their count-controlled interpretation

- *Modular* – the IR shall be separated into a fixed core component providing the means to easily define and manipulate extensions and a set of modular extensions modeling relevant concepts; extensions shall be handled within the intermediate language, not its implementation

- *Compact* – there shall be as few constructs as necessary

Based on those principles we developed and implemented INSPIRE as it is described in the following sections.

Concrete Design Objectives

For the use case of our IR those rather abstract high level design principles are refined to the following concrete design objectives:

- **A Unified Parallel Model** – to provide a *complete*, *explicit* and *unified* representation covering the effects of parallel APIs, a unified parallel model has to be developed and incorporated into the IR. For instance, the various entities processing concurrent control flows, including MPI processes, OpenMP threads, Cilk tasks and OpenCL work-items, need to be modeled utilizing unified constructs since at the level of the Insieme compiler they are all instances of the same fundamental concept. Similarly a concise, coherent formalism for modeling synchronization and communication primitives including barriers, locks and critical regions has to be developed as part of the INSPIRE design.

- **A Concise High-Level IR** – as has been covered earlier, for the given task a high-level, AST like IR is most suitable. However, as has also been pointed out in the introduction, ASTs of actual programming languages cover a huge amount of details and corresponding syntactic subtleties. Many of those features are present for the convenience of human users, including names, nominal type systems, type and access modifiers, function overloading, name resolution procedures, the implicit conversion between R- and L-values, C's interaction between arrays and pointers or C++ references. However, for a compiler IR those extra features are widely dispensable and may therefore be omitted to simplify the development of analysis and transformation tools. Getting rid of as many omissible details as possible to form a *compact*, *simple* an *explicit*, high-level intermediate language has therefore been among the major design objectives.

- **An Open System** – all real-world programs depend on libraries. Those comprise third-party as well as system libraries. To provide *complete* support for general applications our IR should be capable of interfacing with those libraries, especially if they are only available in their binary form such that they can not be processed by the Insieme compiler directly to be incorporated into the program. The IR therefore has to be capable of modeling interfaces of external components. In particular, external functions in C as well as class and template based libraries in C++ have to be accessible. This requirement opposes, and hence constraints, the goal of establishing a concise IR as requested above.

- **Performance** – all these goals have to be achieved by preserving performance critical aspects of the original code. The conversion of an input code into our IR and back must not result in a inherent performance degradation. For instance, a restriction to *single assignments*[1], typical for purely functional languages, could significantly simplify the language design, the implementation of analyzes and the automated parallelization of codes, since expressions no longer have side effects. However, in particular for C codes operating on arrays, an automated conversion into such a restricted format is a complex task and likely to degrade the performance of the original code. Similarly, the utilization of a unified parallel model must not have a negative effect on the performance and scalability of parallel applications. Consequently, the set of features supported by our intermediate language have to be designed such that the performance of input codes is not inherently negatively affected.

The major part of this chapter covers the syntax, semantic and utilization of INSPIRE for C based parallel applications. For brevity the focus is put on C. The few extensions to our IR required to effectively handle C++ and its interfaces are outlined in Section 3.11.

3.3 Overview

Before we dive into the syntactic and semantic aspects of INSPIRE this small section provides an informal overview on the underlying language and its unified parallel model. It also describes how some of the requirements stated in previous section have been approached.

[1]The value of a variable is defined at its initialization and can not be altered afterwards.

3.3.1 Basic Language Design

The design of our IR is subdivided into two components: the *language core* and an extendable set of *language extensions*. Their relation is similar to conventional language definitions consisting of a set of language constructs (core) and definitions of standard libraries (extensions). Hence, the core covers the available set of constructs while extensions are mere utilizations and combinations of those constructs. However, like programming libraries, extensions are utilized to introduce new types and operators without the need to modify the implementation of the core language nor its associated utilities. The separation of those two components therefore provides the foundation for the desired modular and extendable IR design.

The Language Core

The *language core* covers a fixed set of primitives, including type, expression and statement constructs. It has been strongly inspired by functional programming languages and their powerful means for building flexible and reusable functionality. This property is based on supporting functions as first-class citizens, the resulting power of functional composition and a type system including variables enabling the definition of generic implementations of functions. Those concepts have been adapted and integrated into our intermediate language. Additionally, typical type and statement constructs encountered within C-like languages are included to simplify the conversion from and to those languages and to support the interfacing with external libraries.

Another source of inspiration have been formal *specification languages*. Within those languages, algebraic structures consisting of a set of abstract values (a type) and several operators defined on those can be used to specify the behavior of a program or system on a very high level. Within INSPIRE the same principle is used to abstract from implementation details where those may be omitted. To that end, the language core offers means, in particular constructs for abstract types and operators, to define algebraic structures to be processed by our IR. In fact, all primitive types, several basic data structures and external library interfaces are modeled using this language feature.

Additionally, the *language core* specifies rules for the deduction of types, the composition and the evaluation order of expressions, the processing of statements and the scope of variables. The core has been designed to be minimal and simple.

Language Extensions

Based on the core's abilities, *extensions* may be defined. Extensions introduce new types and operators by either using abstract constructs or –

whenever possible – by composing previously defined elements to obtain *derived constructs*. Neither requires the implementation of the IR to be altered. When using abstract constructs, they have to be interpreted correctly by IR utilities including analysis and the backend while derived components may be substituted by their definitions. Consequently, in order to keep the number of cases to be interpreted by utilities low, new abstract constructs should only be introduced if the desired behavior can not be expressed otherwise. In case constructs have to be added, the support for generic types and functions inherited from the functional roots facilitates the coverage of entire families of operators and types using a single construct.

The distinction on whether a certain construct is added to the language core or realized as an extension is based on its semantic. In general, everything that is necessary to describe the (parallel) data and control flow within a program and cannot be described by composing existing constructs is part of the core, while abstract or derived types and operators are to be realized as extensions. However, this definition does not result in a clear boundary between core constructs and extensions. For instance, the while loop statement, definitely influencing the control flow of a program and hence being part of the core languages, is depending on a boolean condition. However, boolean values are abstract values integrated into INSPIRE utilizing an extension. On the other hand, as will be covered in the following syntax section, parallel primitives offered by the core language are realized utilizing the same means as extensions. Still, those constructs belong to the core since they are essential for modeling the parallel control flow of programs.

As will be illustrated in Section 3.7 this distinction provides the foundation for specifying the semantic of the core constructs such that the semantic of extensions can be added in a desired, modular fashion.

Structure

Another significant difference of the Insieme IR when compared to other, conventional IRs is its focus on modeling the program execution, not the structure of the code describing it. Inspired by the simplicity of the self-contained structure of the expressions representing full programs within the *lambda calculus*, we extended this concept to our entire IR. An IR program is just a single expression to be evaluated upon execution. Within this expression functions are incorporated via lambda expressions similar to the way lambda abstractions are utilized within the lambda calculus. Consequently no top-level definitions binding names to functions or types are required. Furthermore, semantically irrelevant constructs like translation units, the distinction between declarations and definitions and the resulting dependencies are effectively eliminated.

Another consequence of this design is that arbitrary sub-expressions of an IR program represent programs on their own. This allows utilities including static code analysis, transformations or the backend to be applied on substructures of complete applications without the need of explicitly isolating them from the rest of the program.

3.3.2 INSPIRE's Unified Parallel Model

Among the most essential contributions in the design of INSPIRE is its unified parallel model. The objective for this component is to provide a common, concise formalism for modeling the effects of parallel APIs as distinct as OpenMP, Cilk, MPI and OpenCL. The resulting model consists five main components:

- *Threads* – basic units of a sequential control flows, processed concurrently with other threads and organized in hierarchical thread groups
- *Thread Groups* – basic units capable of computing parallel control flows, constituting the organizational units for processing jobs
- *Jobs* – parallel units of work being cooperatively processed by thread groups consisting of concurrently running threads
- *Collective Operations* – to distribute workload and data among the members of a thread group
- *Point-to-Point Communication* – to exchange data and to realize synchronization points between individual threads

Based on these primitives and the compositional power of the IR language core, the effects of a variety of constructs offered by parallel APIs can be modeled accurately within the Insieme IR.

Threads, Thread Groups and Jobs

An IR *thread* is an arbitrary entity capable of processing a sequential control flow. This definition covers standard OS-level threads as well as OpenMP threads, Cilk tasks, OpenCL work items and MPI processes. INSPIRE's parallel model does not define any specific relation between its *IR threads*, OS-level threads, processes, HW-threads or other processing entities. This flexibility is exploited by the backend and runtime system, allowing them to implement IR threads using light-weight tasks which are flexibly scheduled on the available computational resources.

Threads can only be created as part of an entire *thread group*. Furthermore, each thread is a member of the one thread group it has been created as a part of. This membership can not be altered afterwards. Thread groups are thus disjoint. A full thread group is the basic unit capable of processing a parallel unit of work within INSPIRE's parallel model. Such a work unit

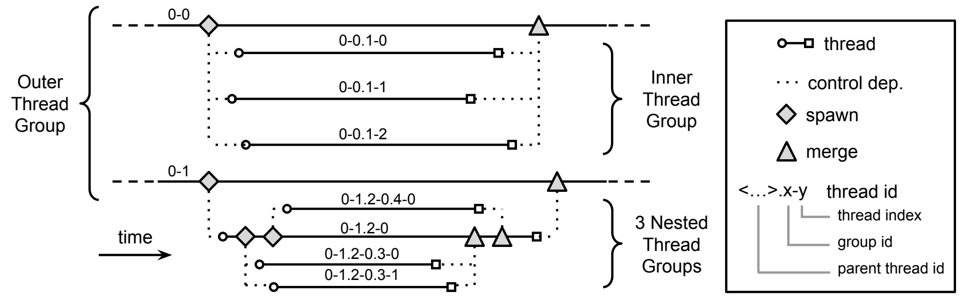


Figure 3.1: Example thread group nesting.

is referred to as a *job*. A job specifies a function call to be executed by each thread of the the processing thread group as well as lower and upper boundaries on the number of involved threads.

The creation of a thread group is realized by a *spawn* operation accepting a job as an argument. It is nondeterministically determining the number of threads to be utilized for processing the given job within the stated boundaries and triggers their execution. The result of the spawn operation is a value referencing the created thread group which later on can be utilized to wait for its completion by invoking a corresponding *merge* operation.

Every thread belongs to exactly one thread group and memberships are static. Furthermore, every thread may spawn nested thread groups which may arbitrary overlap in their execution. Also, the main thread, hence the entry point of an application is considered to be within a thread group consisting of a single thread.

Within a thread group every thread has a unique ID corresponding to its index within the group. Job implementations may query this ID as well as the ID of an arbitrary parent thread (the thread creating the local thread group and its parents) to realize diverging control flows.

Figure 3.1 illustrates an example thread group nesting according to our model. The outermost group consists of two threads (0-0 and 0-1) where the first is spawning a nested group of 3 threads (0-0.1-0, 0-0.1-1 and 0-0.1-2) while the second spawns two additional, overlapping thread groups. Each thread has a numerical identifier including the ID of the parent thread, a unique thread group ID and its index within its group. Threads are not required to merge their child-thread groups on their own, they may forward the group reference to some other thread such that this thread can conduct a merge operation (not shown in Figure 3.1).

This model is sufficient to cover the parallel constructs in our four primary parallel target APIs:

- *OpenMP:* to model a OpenMP parallel region the corresponding code block is outlined into a function call $f(\ldots)$ and a job $job[1,\infty]f(\ldots)$ processing f with an arbitrary number of threads is created. Based on this, the call

$$merge(spawn(job[1, \infty]f(\ldots)))$$

creates a thread group processing the region and waiting for its completion.

- *Cilk:* a Cilk task is just as an OpenMP parallel region outlined into a function call $f(\ldots)$ and encapsulated into a job $j = job[1, 1]f(\ldots)$ restricted to be processed by a thread group consisting of a single thread. The spawn operation is triggered by $g = spawn(j)$ and may be later on merged by processing $merge(g)$.

- *MPI:* the full-program, process level parallelism of MPI can be modeled by replacing the original entry point function main by a new implementation wrapping an invocation of the original main function into a job being processed by an arbitrary number of IR threads similar to the example provided for OpenMP. Hence, MPI processes are modeled utilizing IR threads – just as any other concurrent control flow.

- *OpenCL:* the processing of work items within work item groups can be modeled by creating jobs demanding a corresponding number of threads. For instance, let $f(\ldots)$ be the call describing the processing of a OpenCL work-item. The $job[32, 32]f(\ldots)$ would correspond to a OpenCL work-group consisting of 32 work items which may be processed by a corresponding call to the spawn and merge operators.

The primitives introduced so far provide means for creating and merging thread groups to conduct concurrent operations. However, every non-trivial job being cooperatively processed by a thread group requires the involved thread to coordinate their joined efforts. To do so, communication and synchronization primitives are required.

Inter-Thread Communication

For a thread group to work cooperatively on a parallel job, means for communication are required. Three primitives are offered for this purpose – one enabling the distribution of work ($pfor$), one for distributing and redistributing data throughout a group (*redistribute*) and a third one for point-to-point communication (*channels*). The first two primitives are collective operators, hence in order for them to complete, all threads of a group must participate.

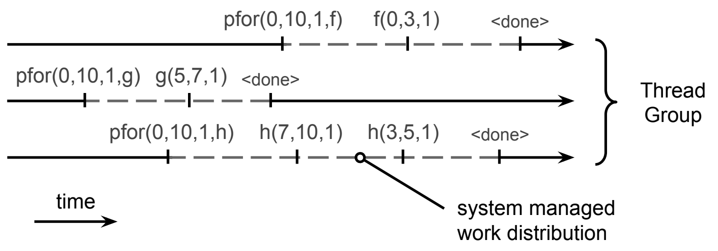


Figure 3.2: Example application of the *pfor* operator.

The Work Distribution Operator Work is distributed using the abstract operator *pfor* which is named after its most prominent use case – the parallel for. An example application is depicted in Figure 3.2. The operator accepts four parameters. The first three define the range of an iterator – start, end and step size. In the illustrated case it is the range $(0, 10, 1) = [0 \dots 9]$. Note that for simplifying splitting operations, the upper boundary is implicitly excluded. All threads within a group have to invoke this operator using the same iterator range. The specified range will be distributed among the available threads using an undefined schema determined by the runtime system. In case a specific scheduling policy should be enforced, it can be encoded directly within the IR. For instance, a *pfor* can be replaced by a *for*-loop processing a share of the total range if a static scheduling should be realized. For dynamic loop scheduling approaches hints supporting the scheduling in the runtime system can be annotated [97].

As its last parameter the *pfor* primitive accepts a function capable of processing sub-ranges of the given range. Each participating thread may pass a different function or closure binding local context information as the fourth parameter. In Figure 3.2 this is indicated by the function symbols f, g and h. The function provided by the local thread is utilized for processing the assigned share of the overall range $(0, 10, 1) = [0 \dots 9]$ within the context of the local IR thread, illustrated by the processing of the calls $f(0, 3, 1)$, $g(5, 7, 1)$, $h(7, 10, 1)$ and $h(3, 5, 1)$ by the corresponding threads. Note that individual threads may get multiple sub-ranges assigned and their processing might be out of order.

After finishing its shares, each thread will continue processing the following statement. There is no implicit barrier at the end. The only guarantee given is that after the last thread has completed the *pfor* call, the entire range has been processed.

The Data Distribution Operator Several parallel models provide primitives to scatter and gather data to and from all participating threads. These kind of operations are particularly prominent in message passing solutions. As for the work-sharing, we aimed to identify a single primitive capable of

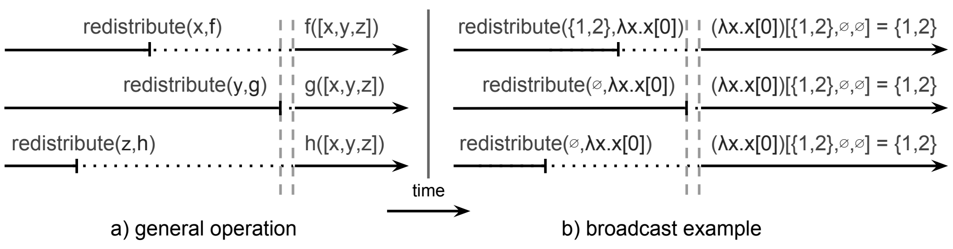


Figure 3.3: Example applications of the *redistribute* operator.

covering all these functionalities. The result is the *redistribute* operation as illustrated in Figure 3.3. The name is based on its capability of taking data computed by individual threads of a group and re-distribute it among those. It accepts two parameters – a value contributed by the local thread and a function describing the extraction of the value to be returned to the local thread from an array aggregating the contributions of all the threads within the thread group.

Note that, despite its name, the *redistribute* operator is a universal operator for collective data exchanges among the members of a group which might as well be utilized for establishing initial data distributions. In such a case one thread would contribute all the data while the remaining threads contribute a token representing no data. The utilized extraction function would then model the desired, initial data distribution.

Part a) of Figure 3.3 illustrates a generic example. Three threads are contributing the data elements x, y and z to the *redistribute* operator which is blocking until the last contribution is delivered. Once available, the contributions are aggregated to the array $[x, y, z]$ and forwarded as an argument to the individual extraction functions passed by the involved threads. The extractions are conducted locally within the contexts of the corresponding threads, as indicated by the calls $f([x, y, z])$, $g([x, y, z])$ and $h([x, y, z])$. Thus, the *redistribute* operator simply collects all the contributions and allows a generic function to select the piece of information to be made available to the local thread.

As an example, part b) of Figure 3.3 illustrates the utilization of the *redistribute* operator for broadcasting the set $\{1, 2\}$ from the first thread to all members of the enclosing thread group. While the first thread is contributing the corresponding set, the others are simply passing an empty set. The function extracting the value for the local thread, however, is identical for all involved threads and it is simply obtaining the value contributed by the first thread[2].

[2]The function $\lambda x.x[0]$ is represented utilizing the syntax of the lambda calculus and extracts the first element of an array passed as an argument.

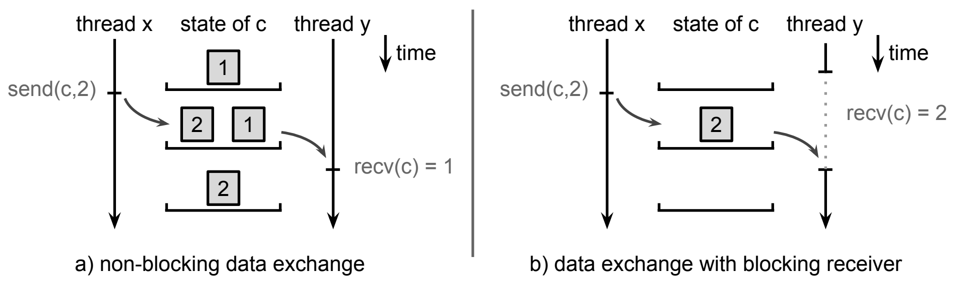


Figure 3.4: Utilization of a channel for inter-thread communication.

Similar to *pfor*, the *redistribute* primitive is a collective operation and needs to be invoked by all threads within a group. However, unlike *pfor* it is a blocking operation.

Implementing collective operations based on an actual realization of the redistribute primitive would result in slow program executions. This internal representation is intended to unify collective operations for analyzes, not to define their implementation. That is why derived constructs have been defined using this central primitive to encapsulate more specialized operations, including *barriers* (in MPI, OpenMP and OpenCL), *reductions* or *broadcasts*. As for most derived operations, their encoding is intercepted in the backend to generate specialized code.

Point-to-Point Communication Finally, a mechanism to send information and synchronization events between individual threads is required. To solve this issue we borrowed the concept of *channels* encountered within model checking utilities [44]. We introduce abstract *channel* types representing bound blocking queues that can be utilized for forwarding information between threads. Synchronized *send* and *recv* operators can be utilized to transfer data through a channel as illustrated in Figure 3.4. The depicted channel state describes the state of the buffer queue associated to the channel instance *c* utilized for the exchange of information. Both operations, the *send* and the *recv*, may block the execution of the calling thread in case the buffer is full (*send*) or empty (*recv*).

Besides transferring data, channels are also used to model locks and bounded semaphores. For instance, a channel with a queue length of 1 is equivalent to a binary semaphore when using the *send* operator as the V and *recv* as the P operator.

Although basic channel operations are always blocking, in combination with *jobs*, *spawn* and *merge* expressions non-blocking *send* and *recv* operations can be realized by wrapping a corresponding basic operation invocation in a job for a single thread which will be spawned and synchronized accordingly. This way the desired asynchronous execution can be achieve.

Communication using Shared Memory INSPIRE does not prohibit the exchange of information using references to shared memory locations. However, as usual, the content of memory locations updated by other threads remains undefined until a synchronization between the source and target thread occurs via a common call to *redistribute* or the forwarding of information using a *channel*.

In general, any memory location may be access by any thread within our representation. There is no concept of ownership or thread private and shared memory. Only the restraint of a thread not to forward a reference to a locally allocated memory location to other threads or the absence of none-thread-local accesses to shared memory locations enables the compiler to map data and control flows to hardware exhibiting limitations on access capabilities, including distributed memory systems, on-device memory of GPUs or scratchpad memory if available.

Note that this behavior models the actual nature of the underlying hardware instead of the semantical constructs offered to the end user simplifying the way to think about and implement parallel programs. For instance, thread private variables in OpenMP are just syntactic sugar covering up for the actual creation of new variables while shared variables are cover-ups of data to be accessed via references or pointers. Consequently, their 'virtual' access restrictions can and may easily be circumvented. Essentially, asserting all data as potentially shared data corresponds to the necessarily conservative point of few to be embodied by a general purpose compiler IR.

Furthermore, the decision of modeling parallel programs as a set of concurrent control flows operating on a pool of shared memory locations isolates INSPIRE from any particular hardware or API dependency, enabling the support of arbitrary language or API constructs and primitives as well as future hardware architectural developments.

3.4 Syntax

In this section the *abstract syntax* of INSPIRE is covered. Its purpose is to formalize the set of constructs and primitives offered to build representations of parallel programs. Further, since in many cases an explicit description of program fragments using the given set of fundamental primitives is overly extensive, abbreviations (syntactic sugar) and conventions are introduced. The result is a *concrete syntax* to be used throughout this thesis that is more intuitive for human readers without losing the concise, formal characteristics of the underlying core primitives.

3.4.1 Core Language Constructs

The language core is constituted by a fixed set of primitives sufficient for modeling all sequential and parallel data and control flows within an application. It further provides means for integrating extensions. The utilization of those means and several essential extensions are covered in Section 3.8.

For the following definitions let $\mathbb{N}$ be the set of natural numbers including 0 and $\mathbb{I}$ be a set of identifiers, e.g. the set of strings over a given alphabet. The abstract syntax of INSPIRE is defined by the following inductive definitions of the sets of types $\mathbb{T}$, expressions $\mathbb{E}$ and statements $\mathbb{S}$.

Types

Definition 3.1 (types)**.** Let $i, i_1, \ldots, i_k \in \mathbb{I}$ be identifiers, $t, t_1, \ldots, t_k \in \mathbb{T}$ be types and $n_1, \ldots, n_k \in \mathbb{I}$ be identifiers utilized as field names for $k \in \mathbb{N}$. Then, for all $x \in \mathbb{N} : 1 \leq x \leq k$,

$$
\begin{aligned}
& i \langle t_1, \ldots, t_k \rangle && \text{(abstract)} \\
& \mathit{struct}\ \{n_1 : t_1, \ldots, n_k : t_k\} && \text{(struct)} \\
& \mathit{union}\ \{n_1 : t_1, \ldots, n_k : t_k\} && \text{(union)} \\
& (t_1, \ldots, t_k) \rightarrow t && \text{(func)} \\
& (t_1, \ldots, t_k) \Rightarrow t && \text{(closure)} \\
& {}'i && \text{(var)} \\
& \mathit{rec}\ {}'i_x . \left\{ {}'i_1 = t_1, \ldots, {}' i_k = t_k \right\} && \text{(rec)} \\
& \mathit{rec}\ {}'i && \text{(rec var)}
\end{aligned}
$$

are types as well. The set $\mathbb{T}$ is the smallest set closed under does constructs.

Note that (abstract), (struct), (union) and (var) constitute base cases for this recursive definition whenever $k = 0$.

The (abstract) rule supports the introduction of *parametrized abstract types*. Primitive types including the boolean type *bool*[3], the signed 4-byte integer $\mathit{int}\langle 4 \rangle$ [4] or the double type $\mathit{real}\langle 8 \rangle$ are represented using this construct. Also basic data structures including dynamically sized arrays of Boolean values ($\mathit{array}\langle \mathit{bool} \rangle$) or statically sized vectors of eight integer values ($\mathit{vector}\langle \mathit{int}\langle 4 \rangle, 8 \rangle$) are based on this type construct and will be covered in more detail in Section 3.8.

The remaining constructs support the construction of struct (struct), union (union), function (func) and closure types (closure). Type variables of the form (var) can be used to define generic types like $(\alpha) \rightarrow \alpha$ which is the type of e.g. the identity function. For an easier distinction to other identifiers Greek letters like α and β are used instead of $'a$ and $'b$. The last two constructs, (rec) and (rec var), provide means for defining recursive types like lists or tree structures. The former binds the recursive type variables $\mathit{rec}\ 'i_1, \ldots, \mathit{rec}\ 'i_k$ to the types $t_1, \ldots, t_k$ to be used within the definition

[3]If $k = 0$ we omit the trailing $\langle\rangle$.

[4]In this context 4 is a abstract type with the identifier $i = 4$ and no parameters.

of the types $t_1, \ldots, t_k$ for establishing recursive relations. The variable $'i_x$ selects the recursive type to be represented by the full construct. The support of multiple bound identifiers extends the expressiveness to mutually recursive types.

Example 3.1 (composed types)**.** To illustrate the utilization of the type system a few C types and their IR equivalents shall be demonstrated. Primitive types like integral or floating point types are covered utilizing the abstract type construct, as has been covered above. So are arrays and vectors. Pointer types may be represented utilizing a parametrized abstract type ref as will be covered in detail in Section 3.9.1. A pointer

```
int*
```

in C will be converted to the type

```
ref<int<4>>
```

within the Insieme IR. Furthermore, a struct type

```
struct {
  int a;
  float b;
}
```

in C is converted to the type

```
struct { a : int<4>, b : real<4> }
```

within the IR, a recursive struct definition like

```
struct list {
  int a; struct list* n;
}
```

is converted to the recursive type

```
rec 'list . {
  'list = struct { a:int<4>, n:ref<rec 'list> }
}
```

and struct A in the mutual recursive definition

```
struct A { struct B* b; };
struct B { struct A* a; };
```

is converted to

```
rec 'A . {
  'A = struct { b : ref<rec 'B> },
  'B = struct { a : ref<rec 'A> }
}
```

and type B in the same definition into

```
rec 'B . {
   'A = struct { b : ref<rec 'B> },
   'B = struct { a : ref<rec 'A> }
}
```

Finally, the C function type

```
int(float, bool)
```

is encoded utilizing the IR type

```
(real<4>,bool)->int<4>
```

There is no C equivalent to closure types as they are supported by our IR. Yet those are required for functional core constructs of our IR. Naturally, type constructs may be arbitrarily nested. However, since no type-definition is offered for naming types, the full structure has to be explicitly nested. Hence, for instance, the function type definition

```
struct A { struct B* b; };
struct B { struct A* a; };
int(float,struct A)
```

will result into

```
(
  real<4>,
  rec 'A . {
     'A = struct { b : ref<rec 'B> },
     'B = struct { a : ref<rec 'A> }
  }
)->int<4>
```

within INSPIRE, demonstrating its structural nature compared to the nominal approach utilized by the C language.

Expressions

Definition 3.2 (variables)**.** Let

$$\mathbb{V} = \{var(i : t) | i \in \mathbb{I} \wedge t \in \mathbb{T}\}$$

be the set of variables where $var(i : t)$ denotes a *variable* $i \in \mathbb{I}$ of type $t \in \mathbb{T}$.

For conveniences we write i instead of $var(i : t)$ whenever its interpretation is clear from the context.

Definition 3.3 (recursive variables)**.** Let

$$\mathbb{T}_{\rightarrow} = \{(t_1, \ldots, t_n) \rightarrow t \mid t_1, \ldots, t_n, t \in \mathbb{T}\}$$

be the set of all function types. Let

$$\mathbb{V}_{rec} = \{rec(i : t) | i \in \mathbb{I} \wedge t \in \mathbb{T}_{\rightarrow}\}$$

be the set of recursive variables where $rec(i : t)$ denotes a *recursive variable* $i \in \mathbb{I}$ of type $t \in \mathbb{T}_{\rightarrow}$.

As for variables we write i instead of $rec(i : t)$ whenever its interpretation is clear from the context.

Definition 3.4 (expressions)**.** Let $i \in \mathbb{I}$ be an identifier, $t_* \in \mathbb{T}$ be types[5], $n_* \in \mathbb{I}$ be names, $e_*, b \in \mathbb{E}$ be expressions, $v_* \in \mathbb{V}$ be variables, $f_* \in \mathbb{V}_{rec}$ be recursive variables and $s_* \in \mathbb{S}$ be statements for $k, l_* \in \mathbb{N}$. Then, for all $x \in \mathbb{N} : 1 \leq x \leq k$,

$$v \tag{var}$$

$$lit(i : t) \tag{lit}$$

$$e(e_1, \ldots, e_k) \tag{call}$$

$$struct\,\{n_1 = e_1, \ldots, n_k = e_k\} \tag{struct}$$

$$union\,\{n = e\} \tag{union}$$

$$e.n \tag{access}$$

$$\begin{aligned} & rec\ f_x.\{ \\ & \qquad f_1 = (v_{11}, \ldots, v_{1l_1}) \rightarrow t_1\ s_1, \\ & \qquad \ldots \\ & \qquad f_k = (v_{k1}, \ldots, v_{kl_k}) \rightarrow t_k\ s_k \\ & \} \end{aligned} \tag{func}$$

$$f \tag{rec}$$

$$(v_1, \ldots, v_k) \Rightarrow e(e_1, \ldots, e_l) \tag{bind}$$

$$job\ [e_l, e_u]\ b \tag{job}$$

are expressions as well. The set $\mathbb{E}$ is the smallest set closed under those constructs.

Note that (var), (lit), (struct), and (rec) constitute base cases for this recursive definition whenever $k = 0$.

Variables (var), *recursive variables* (rec), *literals* (lit) and *call* expressions (call) are required to model the data and control flow, the (access) construct obtains field values from structs or unions and the remaining five enable the construction of struct (struct), union (union), function (func), closure (bind), and job (job) values.

[5]For brevity we use the notation x_* to denote the list of elements x with any or no subscript encountered within the following definition.

Literals provide the means for the introduction of typed constants within the IR. Also, to introduce an abstract function/operator, a constant exhibiting a function type is used. As for variables, we will write $i : t$ or just i instead of the explicit $lit(i : t)$ in cases where the interpretation is clear from the context. Within *call* expressions (call), arguments are eagerly evaluated in an arbitrary order and passed by value. *Lambda* expressions (func) support the definition of (mutual) recursive functions by binding function definitions to *recursive variables* $(f_1, \dots, f_k)$. In case of a non-recursive function $rec\ f\{f = (v_1, \dots, v_k) \rightarrow t\ s\}$, where f does not appear free in s, we omit the definition of f to obtain the equivalent, yet more concise notation $(v_1, \dots, v_k) \rightarrow t\ s$ or even $(v_1, \dots, v_k)\ s$ if the return type is also clear from the context. The bodies $s_1, \dots, s_k$ of the function definitions must not exhibit free variables other than the associated parameters and the variables $f_1, \dots, f_k$. *Bind* expressions (bind) construct a closure value processing a call expression $e(e_1, \dots, e_l)$ upon invocation. Arguments $e_1, \dots, e_l$ which are not parameters of the resulting closure $(v_1, \dots, v_k)$ are captured from the surrounding context. Finally, *job* expressions (job) create *jobs* which are the parallel equivalent of closures. As introduced in Section 3.3.2, jobs are processed asynchronously by groups of threads which form the basic organization unit for collective operations. The full semantic of those constructs is covered in detail in Section 3.7.

Operators and Infix Notation The *abstract syntax* does not cover rules for resolving the precedence order of operators nor support for infix notations. Since the evaluation order of sub-expressions is defined by the recursive composition of expressions in the *abstract syntax* and infix notations are equivalent to call expressions, corresponding constructs would be redundant. However, for the *concrete syntax* we utilize infix notation, parenthesis and common operator precedence rules of the C language family to fix the evaluation order. For instance the term

$$a + b * c$$

is equivalent to

$$int.add(a, int.mul(b, c))$$

where $a, b, c \in \mathbb{V}$ are variables, *int.add* is the literal representing the integer addition operator and *int.mul* the multiplication while

$$(a + b) * c$$

is equivalent to

$$int.mul(int.add(a, b), c)$$

since parenthesis have been utilized to overrule the default evaluation order.

Extended Bind Expression Syntax In the *abstract syntax* the body of a bind expression is strictly limited to a single call expression to simplify the tracing of variable bindings by limiting their scopes. However, as for some other constructs, this restriction results in an overly extensive textual description for the human reader. For the *concrete syntax* we therefore support the utilization of arbitrary expressions and statements as the body. Hence for an arbitrary expression $e \in \mathbb{E}$ the concrete syntax expression

$$(v_1, \ldots, v_k) \Rightarrow e$$

is equivalent to the abstract syntax expression

$$(v_1, \ldots, v_k) \Rightarrow e' \; (v_1, \ldots, v_k, c_1, \ldots, c_l)$$

where e' is the function

$$\begin{array}{l} (v_1, \ldots, v_k, c_1, \ldots, c_l) \rightarrow t \; \{ \\ \qquad \mathit{return} \; e; \\ \} \end{array}$$

and $c_1, \ldots, c_l \in \mathbb{V}$ is the list of captured variables, hence the list of free variables of expression e excluding the parameters $v_1, \ldots, v_k$ of the bind expression. Also $t \in \mathbb{T}$ is the type of expression e. Similarly, for an arbitrary statement $s \in \mathbb{S}$, the concrete syntax expression

$$(v_1, \ldots, v_k) \Rightarrow s$$

is equivalent to the abstract syntax expression

$$(v_1, \ldots, v_k) \Rightarrow e' \; (v_1, \ldots, v_k, c_1, \ldots, c_l)$$

where e' is the function

$$(v_1, \ldots, v_k, c_1, \ldots, c_l) \rightarrow \mathit{unit} \; s$$

and $c_1, \ldots, c_l \in \mathbb{V}$ is the list of captured variables. The abstract generic type *unit* is utilized as the return type of functions that do not produce results but cause desirable side effects and hence, essentially, the type of statements.

Example 3.2 (expressions)**.** As has already been outlined above, most C expressions can be directly converted into IR. For instance, the expression

```
a + a * b
```

is represented e.g. by the IR expression

```
var(a:int<4>) + var(a:int<4>) * var(b:int<4>)
```

which a syntactic sugar based version of

```
int.add(
   var(a:int<4>),
   int.mul(var(a:int<4>),var(b:int<4>))
)
```

where in this example *var(a:int<4>)* corresponds to the C variable a and *var(b:int<4>)* to the variable b. A more complex case is the C function

```
int sum(int a, int b) { return a + b; }
```

which is converted to the expression

```
rec rec(sum:(int<4>,int<4>)→int<4>) . {
   rec(sum:(int<4>,int<4>)→int<4>) =
   (var(a:int<4>),var(b:int<4>))→int<4> {
      return var(a:int<4>) + var(b:int<4>);
   }
}
```

where ***rec****(sum:(int<4>,int<4>)→int<4>)* is the recursive variable closing the function definition and the ***return*** statement is a statement to be introduced in the next sub-section. By abbreviating variables by their identifiers the given code fragment is equivalent to

```
rec sum . {
   sum = (a,b)→int<4> {
      return a + b;
   }
}
```

and since the given function is not recursive we can write

```
(a,b)→int<4> { return a + b; }
```

instead. In constrast, the recursive C function

```
int fac(int n) { return (n==0) ? 1 : fac(n−1)∗n; }
```

is encoded using the IR lambda expression

```
rec fac . {
   fac = (n)→int<4> {
      return (n==0) ? 1 : fac(n−1)∗n;
   }
}
```

where *fac* is the recursive variable ***rec****(fac:(int<4>)→int<4>)*. In this case no contraction to a version not exhibiting the ***rec*** construct is allowed. Finally, the mutual recursive function *even* in

```
bool even(int n) {
   return (n==0) ? true : odd(n−1);
}
```

```
bool odd(int n) {
  return (n==0) ? false : even(n-1);
}
```

is represented by the lambda expression

```
rec even . {
  even = (n)→bool {
    return (n==0) ? true : odd(n-1);
  },
  odd = (n)→bool {
    return (n==0) ? false : even(n-1);
  }
}
```

and the function *odd* by the lambda expression

```
rec odd . {
  even = (n)→bool {
    return (n==0) ? true : odd(n-1);
  },
  odd = (n)→bool {
    return (n==0) ? false : even(n-1);
  }
}
```

As for recursive types, the constructs for representing recursive functions are of a structural nature eliminating the necessity of lookup tables. In particular, all the information required to describe the operation of the function *even* is encoded within its representation.

All expressions have an associated type which can be automatically deduced and checked using inductive type deduction rules (see Section 3.5).

Definition 3.5 (typing statement)**.** The formulation $e : t$ denotes the statement that an expression $e \in \mathbb{E}$ is of type $t \in \mathbb{T}$.

The *typing statement* $e : t$ does not demand that an expression e actually is of type t. The claim may be valid or not. At this point we merely introduce the formalism to denote typing constraints. The associated techniques for checking the validity of typing statements is covered in Section 3.5.

Statements

Definition 3.6 (statements)**.** Let $e_* \in \mathbb{E}$ be expressions, $v \in \mathbb{V}$ be a variable and $s_* \in \mathbb{S}$ be statements. Then, for all $n \in \mathbb{N}$

$$
\begin{aligned}
&e && \text{(expr)}\\
&decl\ v = e && \text{(decl)}\\
&\{s_1; \ldots; s_n\} && \text{(comp)}\\
&if\ \ (e)\ \ then\ s_1\ else\ s_2 && \text{(if)}\\
&while\ \ (e)\ \ do\ s && \text{(while)}\\
&for\ \ (v = e_s \ldots e_e : e_i)\ \ do\ s && \text{(for)}\\
&return\ e && \text{(return)}\\
&break && \text{(break)}\\
&continue && \text{(continue)}
\end{aligned}
$$

are statements as well. The set $\mathbb{S}$ is the smallest set closed under those constructs.

The set of statement constructs reflects typical elements found within imperative languages. The rule (expr) ensures that every expression is a statement. A *variable declaration* (decl) introduces a new variable whose scope is bound statically by the enclosing compound statement (comp). Constructs of the form (if) support the definition of conditionally processed statements. For convenience, within textual notation, the *then* keyword or empty branches may be omitted. For instance, the statement "$if\ (e_c)\ s$" is considered equivalent to "$if\ (e_c)\ then\ s\ else\ \{\}$". Also, the core language offers two distinct loop constructs – the condition-controlled *while* loop and the count-controlled *for* loop. In the latter the step size may be omitted if it is equivalent to 1.

Let Bindings

In many cases it might be necessary to repeat complex IR sub-structures several times within code fragments. To avoid redundant formulations within *concrete syntax* let bindings are introduced.

Definition 3.7 (substitution)**.** Let $c, c_1, c_2 \in \mathbb{T} \cup \mathbb{E} \cup \mathbb{S}$ be IR fragments. Then

$$c\,[c_1/c_2] \in \mathbb{T} \cup \mathbb{E} \cup \mathbb{S}$$

denotes the IR fragment obtained by substitution every occurrence of c_2 within c by c_1.

Definition 3.8 (let bindings)**.** Let $i \in \mathbb{I}$ be an identifier an $t_1, t_2 \in \mathbb{T}$ be types. Then

$$let\ i = t_1\ in\ t_2$$

is equivalent to $t_2\,[t_1/i] \in \mathbb{T}$, hence the type obtained by substituting every occurrence of i (abstract type) within t_2 by t_1. Similar, let $e_1, e_2 \in \mathbb{E}$ be expressions and $t \in \mathbb{T}$ be e_1's type (hence $e_1 : t$ is valid). Then

$$let\ i = e_1\ in\ e_2$$

is equivalent to $e_2\,[e_1/lit(i : t)] \in \mathbb{E}$.

Definition 3.9 (let statements)**.** Let $\mathbb{L} = \{let\ i = x | i \in \mathbb{I} \wedge x \in \mathbb{T} \cup \mathbb{E}\}$ be the set of let-statements, $t \in \mathbb{T}$ be a type, $e \in \mathbb{E}$ be an expression of type $t_e \in \mathbb{E}$ (hence $e : t_e$), $a_1, \ldots, a_l \in \mathbb{S} \cup \mathbb{L}$ be statements or let-statements and $b_1, \ldots, b_k \in \mathbb{S}$ be statements. Then for all $l, k \in \mathbb{N}$, the code fragment

$$\{a_1; \ldots, a_l; let\ i = t; b_1; \ldots; b_k\}$$

is equivalent to

$$\{a_1; \ldots, a_l; b_1[t/i]; \ldots; b_k[t/i]\}$$

and the code fragment

$$\{a_1; \ldots, a_l; let\ i = e; b_1; \ldots; b_k\}$$

is equivalent to

$$\{a_1; \ldots, a_l; b_1[e/lit(i : t_e)]; \ldots; b_k[e/lit(i : t_e)]\}$$

Note that *let bindings* and *let statements* are merely syntactic sugar to improve the readability of the IR for the human user. They do not extend the expressiveness of INSPIRE. In particular, let bindings and statements are not type, expression or statement constructs. By definition both, let bindings and statements, are evaluated from the innermost to the outermost level, from the right to the left. Also, let statements are limited to the scope defined by the surrounding compound statement.

3.4.2 Parallel Primitives

In addition to the type, expression and statement constructs introduced so far, the core language comprises a set of primitives orchestrating the parallel execution of applications. All of them, with the exception of the *job expression* (Definition 3.4), are defined using *abstract types* and *literals* (Definitions 3.1 and 3.4) since they neither influence the type system nor the syntactical composition of INSPIRE codes. Nevertheless, they are essential for modeling the parallel control and data flow within applications and hence they are an integral part of the core language.

Definition 3.10 (jobs)**.** Let $e_l, e_u \in \mathbb{E}$ be expressions of type $uint\,\langle 4\rangle$ and $b \in \mathbb{E}$ be a function of type $() \Rightarrow unit$. The job expression

$$job\,[e_l, e_u]\ \ b$$

produces a value of the abstract type *job*.

The function b specifies the operations to be conducted by each of the threads of a thread group when processing the resulting job. The expressions e_l and e_u define lower and upper boundaries for the number of threads required for processing the job. The full details of the semantic are covered by Section 3.7.2.

All threads within a group evaluate the call $b()$, hence they are processing the same sequential code. Furthermore, all threads in a group are indexed. Index dependent control flow can be used to cause execution traces to diverge.

Definition 3.11 (thread identification)**.** The abstract functions

$$getThreadID : (uint\,\langle 4\rangle) \to uint\,\langle 4\rangle$$
$$getNumThreads : (uint\,\langle 4\rangle) \to uint\,\langle 4\rangle$$

retrieve the thread index and the thread group size from within a thread.

The accepted parameter of type $uint\,\langle 4\rangle$ (4-byte unsigned integer) determines the nesting level. Passing 0 returns the index and size of the local group, passing 1 the corresponding values of the parent group and so forth.

Definition 3.12 (thread group management)**.** Every thread is allowed to create nested thread groups using the function

$$spawn : (job) \to thread\_group$$

The resulting value of type *thread_group* references the newly started group. The function

$$merge : (thread\_group) \to unit$$

blocks until the referenced group has completed its job. In case several thread groups have been spawned, a call to

$$merge : () \to unit$$

can be used to wait for the completion of all groups directly spawned by the processing thread.

The Work Distribution Operator

For a thread group to work cooperatively on a parallel job, means for communication are required. Three primitives are offered for this purpose – one enabling the distribution of work ($pfor$), one for redistributing data throughout a group (*redistribute*) and a third one for point-to-point communication (*channels*). The first two primitives are collective operators, hence in order for them to complete, all threads of a group must participate.

Definition 3.13 (work distribution operator)**.** Let $int = int\,\langle 4 \rangle$. The abstract operator

$$pfor : (int, int, int, (int, int, int) \Rightarrow unit) \rightarrow unit$$

is distributing work among the members of the local thread group.

The operator is named after its most prominent use case – the parallel for. The first three parameters define the range of an iterator (start, end and step size) while the last parameter accepts a function capable of processing sub-ranges of the given range. All threads within a group have to invoke this operator using identical values for the first three arguments. The specified range will be distributed among the available threads using a schema that remains undefined within the static program representation. Only during the actual execution a work load distribution will be determined by the dynamic optimizer of the runtime system. In case a specific scheduling policy should be enforced, it can be encoded directly within the IR. For instance, a $pfor$ can be replaced by a for-loop processing a share of the total range if a static scheduling should be modeled.

After finished its shares, each thread will continue processing the following statement. There is no implicit barrier at the end. The only guarantee given is that after the last thread has completed the $pfor$ call, the entire range has been processed (see Section 3.7.2).

The Data Distribution Operator

Several parallel models provide primitives to scatter and gather data to and from all participating threads. These kind of operations are particularly prominent in message passing solutions. As for the work-sharing, we aimed to identify a single primitive capable of covering all these functionality.

Definition 3.14 (data distribution operator)**.** The abstract generic operator

$$redistribute : (\alpha, (array\,\langle \alpha \rangle) \Rightarrow \beta) \rightarrow \beta$$

is (re-)distributing data among the members of the local thread group.

Similarly to $pfor$, this primitive is a collective operation and needs to be invoked by all threads within a group. However, unlike $pfor$ the redistribute primitive is a blocking operation. Every thread contributes a piece of information via the first parameter of type α. The data of all threads is aggregated to an array and passed as an argument to the function specified by the second parameter[6]. The result of the evaluation of this function is returned as the result of the call to the *redistribute* primitive. Thus, the operator simply collects all the contributions and allows a generic function to select the piece of information to be made available to the local thread.

[6]The parametrized abstract type family $array\,\langle \ldots \rangle$ is used within INSPIRE to model arrays – see Section 3.8.2.

Point-to-Point Communication

Finally, a mechanism to send information between individual threads is required. To solve this issue we borrowed the concept of *channels* as used by model checking utilities [44].

Definition 3.15 (channels and channel operators)**.** Let $t \in \mathbb{T}$ be a type and $a \in \{x \in \mathbb{T} \mid x =' i \vee (x = i\langle\rangle \wedge i \in \mathbb{N})\}$ be a numerical type parameter. A value of the parametrized abstract type $channel\langle t, a\rangle \in \mathbb{T}$ is referencing a *channel* transferring values of type t in-order between threads utilizing a buffer queue of size a. Channels are created using the generic abstract operator

$$channel.create : (type\langle\alpha\rangle, param\langle\beta\rangle) \to channel\langle\alpha,\beta\rangle$$

and released using

$$channel.release : (channel\langle\alpha,\beta\rangle) \to unit$$

where $type\langle\alpha\rangle$ is a meta-type representing arbitrary types and $param\langle\beta\rangle$ a meta type for types representing integer constants, e.g. the abstract generic type $4 = 4\langle\rangle \in \mathbb{T}$ (see Section 3.8.2). Furthermore, the operator

$$channel.send : (channel\langle\alpha,\beta\rangle, \alpha) \to unit$$

inserts a new value into the buffer queue while

$$channel.recv : (channel\langle\alpha,\beta\rangle) \to \alpha$$

takes values out of the buffer. Both operations may block the execution of the calling thread in case the buffer is full (*send*) or empty (*recv*). The non-blocking operators

$$\begin{gathered} channel.empty : (channel\langle\alpha,\beta\rangle) \to bool \\ channel.full : (channel\langle\alpha,\beta\rangle) \to bool \end{gathered}$$

can be utilized to probe the current state of a channel.

For a firm description of the semantic of channel operations interested readers are referred to Section 3.7.2 covering a detailed specification of the channel operators' semantic.

3.5 The Type System

An essential part of the INSPIRE language specification is contributed by its type system. As within most high-level programming languages the type system is utilized to verify the proper composition of language constructs.

By providing support for explicit recursive types, type variables, sub-typing relations and parametrized types, the type system of our IR incorporates several features exceeding those of comparable systems encountered within other high-level languages, including those of C/C++ or Java.

The set of types $\mathbb{T}$ and the structure of its elements has been given within Definition 3.1. The following definitions are based on those and have been inspire by Benjamin Pierce's "Types and Programming Languages" [83].

3.5.1 Domains

The type of an expression is a restriction on the value the expression is evaluated to during execution. The set of allowed values covered by a type is its *domain.* By defining the *domain* of a type its semantic interpretation is specified.

Our IR's type system is based on abstract base types and a variety of *type constructors.* The effect of those constructs is defined by the domains they are creating based on the domains of their arguments. However, not every type $t \in \mathbb{T}$ has an associated domain. For instance, the type variable $\alpha \in \mathbb{T}$ is an IR type without an associated domain. A domain for a type variable can only be determined by considering the context it is utilized in. The subset of types a domain can be assigned to is the set of *closed types.* To define this set we first have to provide a definition for *free type variables.*

Definition 3.16 (free type variables)**.** Let $\mathbb{T}_{rvar} = \{t \in \mathbb{T} \mid t = rec\ 'i\} \subset \mathbb{T}$ be the set of *recursive type variables.* The set $F(t) \subset \mathbb{T}$ of *free type variables* of a type $t \in \mathbb{T}$ is defined by

$$F(t) := \begin{cases} \bigcup_{i=1}^{k} F(t_i) & \text{if } t = i\,\langle t_1, \ldots, t_k \rangle \\ \bigcup_{i=1}^{k} F(t_i) & \text{if } t = struct\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \bigcup_{i=1}^{k} F(t_i) & \text{if } t = union\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \bigcup_{j=0}^{k} F(t_j) \cap \mathbb{T}_{rvar} & \text{if } t = (t_1, \ldots, t_k) \to t_0 \\ \bigcup_{j=0}^{k} F(t_j) \cap \mathbb{T}_{rvar} & \text{if } t = (t_1, \ldots, t_k) \Rightarrow t_0 \\ \{'i\} & \text{if } t =' i \\ \bigcup_{i=1}^{k} F(t_i) \setminus \bigcup_{j=1}^{k} \{rec\ 'i_j\} & \text{if } t = rec\ 'i_x.\{'i_1 = t_1, \ldots,' i_k = t_k\} \\ \{rec\ 'i\} & \text{if } t = rec\ 'i \end{cases}$$

Within the INSPIRE type system, two kinds of type variables are present – variables forming placeholders for arbitrary types and recursive type variables utilized for constructing recursive types. Recursive type variables are bound by the most closely nested recursive type construct providing a definition for it while ordinary type variables are bound by the outermost function or closure type construct.

Definition 3.17 (closed types)**.** A type $t \in \mathbb{T}$ is a *closed type* iff $F(t) = \emptyset$. The set of all closed types is denoted by $\mathbb{T}_c \subset \mathbb{T}$.

Definition 3.18 (generic types)**.** A type $t \in \mathbb{T}$ is a *generic type* iff $F(t) \neq \emptyset$ and $F(t) \cap \mathbb{T}_{rvar} = \emptyset$. The set of all generic types is denoted by $\mathbb{T}_g \subset \mathbb{T}$.

Definition 3.19 (valid types)**.** The set $\mathbb{T}_v = \mathbb{T}_c \cup \mathbb{T}_g$ is the set of all valid types.

Only valid types may occur within INSPIRE code. The main restriction of valid types compared to the general set of types is the limitation to types that do not exhibit free recursive type variables.

Definition 3.20 (generic type variables)**.** Let $\mathbb{T}_{gvar} = \{x \in \mathbb{T} \mid x =' i\}$ be the set of all *generic type variables.* The set $G(t) \subset \mathbb{T}_{gvar}$ of *generic type variables* of a type $t \in \mathbb{T}$ is defined by

$$G(t) := \begin{cases} \bigcup_{i=1}^{k} G(t_i) & \text{if } t = i\,\langle t_1, \ldots, t_k\rangle \\ \bigcup_{i=1}^{k} G(t_i) & \text{if } t = \mathit{struct}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \bigcup_{i=1}^{k} G(t_i) & \text{if } t = \mathit{union}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \bigcup_{j=0}^{k} G(t_j) & \text{if } t = (t_1, \ldots, t_k) \to t_0 \\ \bigcup_{j=0}^{k} G(t_j) & \text{if } t = (t_1, \ldots, t_k) \Rightarrow t_0 \\ \{'i\} & \text{if } t =' i \\ \bigcup_{i=1}^{k} G(t_i) & \text{if } t = \mathit{rec}\ 'i_x.\{'i_1 = t_1, \ldots,' i_k = t_k\} \\ \emptyset & \text{if } t = \mathit{rec}\ 'i \end{cases}$$

Note that, unlike within the definition of *free type variables*, recursive type variables are not included within the sets of *generic type variables.*

In the following steps we provide definitions for the instantiations of generic and recursive type variables. Both of them are based on *partial mappings* to be defined first.

Definition 3.21 (partial mapping)**.** A *partial mapping* (or just *mapping*) between a key set K and a value set V is a term of the grammar

$$m ::= \epsilon \mid m[k \mapsto v]$$

where ϵ is the empty mapping, $k \in K$ is an arbitrary key element and $v \in V$ is the value k is bound to. The set of all mappings from a set K to a set V is denoted by $K \rightharpoonup V$. Further, let $m \in (K \rightharpoonup V)$ be a mapping and $x \in K$ be an arbitrary key element. The lookup operation $m[x]$ is defined by

$$m[x] = \begin{cases} v & \text{if } m = m_1[x \mapsto v] \\ m_1[x] & \text{if } m = m_1[y \mapsto v] \text{ and } y \neq x \\ \text{undefined} & \text{if } m = \epsilon \end{cases}$$

Also, the *domain* $\mathrm{dom}(m) \subseteq K$ of a mapping $m \in (K \rightharpoonup V)$ is defined by

$$\mathrm{dom}(m) = \begin{cases} \emptyset & \text{if } m = \epsilon \\ \{x\} \cup \mathrm{dom}(m_1) & \text{if } m = m_1[x \mapsto v] \end{cases}$$

and the concatenation of two mappings a and b denoted by ab is given by

$$ab = \begin{cases} a & \text{if } b = \epsilon \\ ab_1[x \mapsto v] & \text{if } b = b_1[x \mapsto v] \end{cases}$$

Further let $L \subseteq K$ be a set of keys. The operation $m \setminus L$ removes all bindings of keys present in L from m and is defined by

$$m \setminus L = \begin{cases} \epsilon & \text{if } m = \epsilon \\ (m_1 \setminus L)[x \mapsto v] & \text{if } m = m_1[x \mapsto v] \text{ and } x \notin L \\ m_1 \setminus L & \text{if } m = m_1[x \mapsto v] \text{ and } x \in L \end{cases}$$

Finally the removal of a single binding for key $x \in K$ from a mapping m is denoted by $m \setminus x$ and equivalent to $m \setminus \{x\}$.

If the interpretation is clear from the context we write $[a \mapsto b][c \mapsto d]$ or even $[a \mapsto b, c \mapsto d]$ as an abbreviation of the mapping $\epsilon[a \mapsto b][c \mapsto d]$.

Definition 3.22 (generic type variable instantiation)**.** Let $\mathbb{T}_{gvar} = \{x \in \mathbb{T} \mid x =' i\}$ be the set of all *generic type variables.* A *(generic) type variable instantiation* σ is a partial mapping $\mathbb{T}_{gvar} \rightharpoonup \mathbb{T}$ that maps generic type variables to types. Let $t \in \mathbb{T}$ be a type. The instantiated type $\sigma(t) \in \mathbb{T}$ obtained by applying the variable instantiation σ on t is defined by

$$\sigma(t) :=$$

$$\begin{cases} i\,\langle \sigma(t_1), \dots, \sigma(t_k) \rangle & \text{if } t = i\,\langle t_1, \dots, t_k \rangle \\ \mathit{struct}\ \{n_1 : \sigma(t_1), \dots, n_k : \sigma(t_k)\} & \text{if } t = \mathit{struct}\ \{n_1 : t_1, \dots, n_k : t_k\} \\ \mathit{union}\ \{n_1 : \sigma(t_1), \dots, n_k : \sigma(t_k)\} & \text{if } t = \mathit{union}\ \{n_1 : t_1, \dots, n_k : t_k\} \\ (\sigma(t_1), \dots, \sigma(t_k)) \rightarrow \sigma(t_0) & \text{if } t = (t_1, \dots, t_k) \rightarrow t_0 \\ (\sigma(t_1), \dots, \sigma(t_k)) \Rightarrow \sigma(t_0) & \text{if } t = (t_1, \dots, t_k) \Rightarrow t_0 \\ 'i & \text{if } t =' i \text{ and } 'i \notin \mathrm{dom}(\sigma) \\ \sigma['i] & \text{if } t =' i \text{ and } 'i \in \mathrm{dom}(\sigma) \\ \mathit{rec}\ 'i.\, \{'i_1 = \sigma(t_1) \dots ' i_k = \sigma(t_k)\} & \text{if } t = \mathit{rec}\ 'i.\, \{'i_1 = t_1 \dots ' i_k = t_k\} \\ \mathit{rec}\ 'i & \text{if } t = \mathit{rec}\ 'i \end{cases}$$

Let $\sigma, \tau \in \mathbb{T}_{gvar} \rightharpoonup \mathbb{T}$ be two type variable instantiations. The application of σ on τ, denoted by $\sigma(\tau)$, is defined by

$$\sigma(\tau) = \begin{cases} \epsilon & \text{if } \tau = \epsilon \\ \sigma(\tau_1)[v \mapsto \sigma(t)] & \text{if } \tau = \tau_1[v \mapsto t] \end{cases}$$

Furthermore, the composition of σ and τ, denoted by $\sigma \circ \tau$, is given by

$$\sigma \circ \tau = (\sigma(\tau))(\sigma \setminus (\mathrm{dom}(\sigma) \cap \mathrm{dom}(\tau)))$$

hence, the concatenation of $\sigma(\tau)$ and $(\sigma \setminus (\text{dom}(\sigma) \cap \text{dom}(\tau)))$ where the latter covers those variables which are bound by σ but not referenced by τ. The composition $\sigma \circ \tau$ describes the effective variable instantiation applied when consecutively applying σ after τ on a given type. Hence, for any type $t \in \mathbb{T}$ we have $(\sigma \circ \tau)(t) = \sigma(\tau(t))$.

Example 3.3 (variable instantiations)**.** Let

$$\sigma = \epsilon[\alpha \mapsto t_1][\beta \mapsto t_2]$$

and

$$\tau = \epsilon[\alpha \mapsto t_3][\gamma \mapsto A\langle\alpha\rangle]$$

be two variable instantiations. Their domains are $\text{dom}(\sigma) = \{\alpha, \beta\}$ and $\text{dom}(\tau) = \{\alpha, \gamma\}$ respectively. Being applied on a type $B\langle\alpha, \beta, \gamma\rangle$ we obtain

$$\sigma(B\langle\alpha, \beta, \gamma\rangle) = B\langle t_1, t_2, \gamma\rangle$$

and

$$\tau(B\langle\alpha, \beta, \gamma\rangle) = B\langle t_3, \beta, A\langle\alpha\rangle\rangle$$

Further the composition $\sigma \circ \tau$ is given by

$$\begin{aligned}
\sigma \circ \tau &= \sigma(\tau)(\sigma \setminus \text{dom}(\sigma) \cap \text{dom}(\tau)) \\
&= (\epsilon[\alpha \mapsto \sigma(t_3)][\gamma \mapsto \sigma(A\langle\alpha\rangle)])(\sigma \setminus \{\alpha, \beta\} \cap \{\alpha, \gamma\}) \\
&= (\epsilon[\alpha \mapsto t_3][\gamma \mapsto A\langle t_1\rangle])(\epsilon[\alpha \mapsto t_1][\beta \mapsto t_2] \setminus \{\alpha\}) \\
&= (\epsilon[\alpha \mapsto t_3][\gamma \mapsto A\langle t_1\rangle])(\epsilon[\beta \mapsto t_2]) \\
&= \epsilon[\alpha \mapsto t_3][\gamma \mapsto A\langle t_1\rangle][\beta \mapsto t_2]
\end{aligned}$$

Being applied to $B\langle\alpha, \beta, \gamma\rangle$ we obtain

$$(\sigma \circ \tau)(B\langle\alpha, \beta, \gamma\rangle) = B\langle t_3, t_2, A\langle t_1\rangle\rangle$$

which is equivalent to

$$\sigma(\tau(B\langle\alpha, \beta, \gamma\rangle)) = B\langle t_3, \beta, A\langle\alpha\rangle\rangle = B\langle t_3, t_2, A\langle t_1\rangle\rangle$$

as desired.

Definition 3.23 (concrete type instantiation)**.** The set $I(t) \subset \mathbb{T}_c$ of *concrete type instantiations* is defined by

$$I(t) = \{x \in \mathbb{T} \mid G(x) = \emptyset \wedge \exists\sigma.\sigma(t) = x\}$$

Definition 3.24 (recursive type unfolding and folding)**.** Let $\mathbb{T}_{rvar} = \{t \in \mathbb{T} \mid t = rec\,'i\} \subset \mathbb{T}$ be the set of *recursive type variables* and σ_r be a partial mapping $\mathbb{T}_{rvar} \rightharpoonup \mathbb{T}$. Let $t \in \mathbb{T}$ be a type. The type $\sigma_r(t) \in \mathbb{T}$ obtained by applying the recursive variable instantiation σ_r on t is defined by

$$\sigma_r(t) := \begin{cases} i\,\langle \sigma_r(t_1), \ldots, \sigma_r(t_k)\rangle & \text{if } t = i\,\langle t_1, \ldots, t_k\rangle \\ \\ \mathit{struct}\ \{n_1 : \sigma_r(t_1), \ldots, n_k : \sigma_r(t_k)\} & \\ & \text{if } t = \mathit{struct}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \\ \mathit{union}\ \{n_1 : \sigma_r(t_1), \ldots, n_k : \sigma_r(t_k)\} & \\ & \text{if } t = \mathit{union}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\ \\ (\sigma_r(t_1), \ldots, \sigma_r(t_k)) \to \sigma_r(t_0) & \\ & \text{if } t = (t_1, \ldots, t_k) \to t_0 \\ \\ (\sigma_r(t_1), \ldots, \sigma_r(t_k)) \Rightarrow \sigma_r(t_0) & \\ & \text{if } t = (t_1, \ldots, t_k) \Rightarrow t_0 \\ \\ {}'i & \text{if } t =' i \\ \\ \mathit{rec}\ {}'i_x.\{{}'i_1 = \sigma_r'(t_1), \ldots,' i_k = \sigma_r'(t_k)\} & \\ & \text{if } t = \mathit{rec}\ {}'i_x.\{{}'i_1 = t_1, \ldots,' i_k = t_k\} \wedge \\ & \quad \sigma_r' = \sigma_r \setminus \{\mathit{rec}\ {}'i_1, \ldots, \mathit{rec}\ {}'i_k\} \\ \\ \mathit{rec}\ {}'i & \text{if } t = \mathit{rec}\ {}'i \wedge t \notin \mathrm{dom}(\sigma_r) \\ \\ \sigma_r[\mathit{rec}\ {}'i] & \text{if } t = \mathit{rec}\ {}'i \wedge t \in \mathrm{dom}(\sigma_r) \end{cases}$$

The partial function unfold : $\mathbb{T} \to \mathbb{T}$ is *unfolding* recursive types and defined by

$$\begin{aligned} \mathrm{unfold}(\mathit{rec}\ {}'i_x.\, \{{}'i_1 = t_1, \ldots,' i_k = t_k\}) := & \\ & \epsilon[\mathit{rec}\ {}'i_1 \mapsto \mathit{rec}\ {}'i_1.\, \{{}'i_1 = t_1, \ldots,' i_k = t_k\}] \\ & \ [\mathit{rec}\ {}'i_2 \mapsto \mathit{rec}\ {}'i_2.\, \{{}'i_1 = t_1, \ldots,' i_k = t_k\}] \\ & \ \ldots \\ & \ [\mathit{rec}\ {}'i_k \mapsto \mathit{rec}\ {}'i_k.\, \{{}'i_1 = t_1, \ldots,' i_k = t_k\}](t_x) \end{aligned}$$

and the partial function fold : $\mathbb{T} \to \mathbb{T}$ is defined as the inverse operation of unfold.

Note that neither unfold nor fold are total. While unfold is only defined on recursive types, fold is restricted to types which are the result of an unfold operation being applied to a recursive type.

The following definition is fixing the domains of types and hence the interpretation of the type constructors constituting the INSPIRE type system.

Definition 3.25 (type domains)**.** Let D_a be the domain of any closed abstract type $a \in \mathbb{T}_c$ fixed by the language extension introducing the abstract type a. Further, for two sets A and B let $A \to B$ denote the set of all operations[7] that accept an element of A as an argument and compute a value of set B.

The *domain* $D(t)$ of a closed type $t \in \mathbb{T}_c$ is defined by the solution of the set of equations generated by $\mathcal{D}(t)$ which is defined by

$$\mathcal{D}(t) := \begin{cases} \{D(t) = D_{i\langle t_1,\ldots,t_k\rangle}\} \quad \text{if } t = i\,\langle t_1,\ldots,t_k\rangle \\[1em] \{D(t) = D(t_1) \times \ldots \times D(t_k)\} \cup \bigcup_{i=1}^{k} \mathcal{D}(t_i) \\ \qquad\qquad\qquad\qquad \text{if } t = \mathit{struct}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\[1em] \{D(t) = \bigcup_{i=1}^{k} D(t_i)\} \cup \bigcup_{i=1}^{k} \mathcal{D}(t_i) \\ \qquad\qquad\qquad\qquad \text{if } t = \mathit{union}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\[1em] \{D(t) = (D(t_1) \times \ldots \times D(t_k)) \to D(t_0)\} \cup \bigcup_{i=0}^{k} \mathcal{D}(t_i) \\ \qquad\qquad\qquad\qquad \text{if } t = (t_1, \ldots, t_k) \to t_0 \wedge G(t) = \emptyset \\[1em] \{D(t) = \bigcup_{i \in I(t)} D(i)\} \cup \bigcup_{i \in I(t)} \mathcal{D}(i) \\ \qquad\qquad\qquad\qquad \text{if } t = (t_1, \ldots, t_k) \to t_0 \wedge G(t) \neq \emptyset \\[1em] \{D(t) = D((t_1, \ldots, t_k) \to t_0)\} \cup \mathcal{D}((t_1, \ldots, t_k) \to t_0) \\ \qquad\qquad\qquad\qquad \text{if } t = (t_1, \ldots, t_k) \Rightarrow t_0 \\[1em] \{D(t) = D(\text{unfold}(t))\} \cup \mathcal{D}(\text{unfold}(t)) \\ \qquad\qquad\qquad\qquad \text{if } t = \mathit{rec}\ 'i_x.\{'i_1 = t_1, \ldots, 'i_k = t_k\} \end{cases}$$

Note that the cases of type variables and recursive type variables can be omitted since types of this shape are neither closed nor reached by the given recursive definition.

It can also be observed that the domains of a function type $(a) \to b$ and a closure type $(a) \Rightarrow b$ for some types $a, b \in \mathbb{T}$ are equivalent. This is based on the fact that those values are only distinguished by their origin within an application. While values of the function type are introduced by the definition of *function expressions* closure types are created by *bind expressions*. The origin is not required to be distinguished when handling functions within INSPIRE, yet when interfacing with external C/C++ APIs only values of the *function type* $(a) \to b$ can be forwarded since those languages do not support closures.

[7]We use the term *operation* to avoid confusion with (mathematical) functions which are equivalent to *pure functions* within programming languages – operations may have side effects.

Example 3.4 (domains)**.** Let *int* be an integer type such that $D_{int} = \mathbb{Z}$ and the generic type $ref\ \langle\alpha\rangle$ be a reference type such that for all $t \in \mathbb{T}_c$ we have $D_{ref\langle t\rangle} = R$ where R is a arbitrary set of references. Further, let t_r be the recursive type

$$t_r = rec\ \alpha.\{\alpha = struct\ \ \{a : int, b : ref\ \langle rec\ \alpha\rangle\}\}$$

and

$$\begin{aligned} t_s &= \text{unfold}(t_r) \\ &= struct\ \ \{a : int, b : ref\ \langle t_r\rangle\} \\ &= struct\ \ \{a : int, b : ref\ \langle rec\ \alpha.\{\alpha = struct\ \ \{a : int, b : ref\ \langle rec\ \alpha\rangle\}\}\rangle\} \end{aligned}$$

Note that all those types are closed types. To compute the domain $D(t_r)$ the set of equations $\mathcal{D}(t_r)$ has to be obtained and solved. We therefore compute the set of constraints

$$\begin{aligned} \mathcal{D}(t_r) &= \{D(t_r) = D(\text{unfold}(t_r))\} \cup \mathcal{D}(\text{unfold}(t_r)) \\ &= \{D(t_r) = D(t_s)\} \cup \mathcal{D}(t_s) \\ &= \{D(t_r) = D(t_s)\} \cup \{D(t_s) = D(int) \times D(ref\ \langle t_r\rangle)\} \\ &\qquad \cup \mathcal{D}(int) \cup \mathcal{D}(ref\ \langle t_r\rangle) \\ &= \{D(t_r) = D(t_s)\} \cup \{D(t_s) = D(int) \times D(ref\ \langle t_r\rangle)\} \\ &\qquad \cup \{D(int) = D_{int}\} \cup \{D(ref\ \langle t_r\rangle) = D_{ref\langle t_r\rangle}\} \\ &= \{D(t_r) = D(t_s)\} \cup \{D(t_s) = D(int) \times D(ref\ \langle t_r\rangle)\} \\ &\qquad \cup \{D(int) = \mathbb{Z}\} \cup \{D(ref\ \langle t_r\rangle) = R\} \\ &= \{D(t_r) = D(t_s), D(t_s) = D(int) \times D(ref\ \langle t_r\rangle), \\ &\qquad D(int) = \mathbb{Z}, D(ref\ \langle t_r\rangle) = R\} \end{aligned}$$

and solve it for $D(t_r)$ such that we obtain

$$D(t_r) = D(t_s) = D(int) \times D(ref\ \langle t_r\rangle) = \mathbb{Z} \times R$$

which is the domain of t_r.

3.5.2 Type Relations

Before defining rules for deducing types for expressions two auxiliary relations between types have to be defined.

Definitionally Equal

Whenever two types $t_1, t_2 \in \mathbb{T}$ are (structurally) equal ($t_1 = t_2$) the represented type and its associated domain is identical. However the reverse does not hold since two structurally different types can describe the same domain of values. For instance, the two function types $(\alpha) \rightarrow \alpha$ and $(\beta) \rightarrow \beta$ describe the same set of operators – operators accepting an argument of some type and returning a value of the same type – yet the types are obviously not structurally equal. A similar observation can be made regarding recursive types. The domains of the types

$$rec\ \alpha.\{\alpha = struct\{next : \alpha\}\}$$

and

$$struct\{next : rec\ \alpha.\{\alpha = struct\{next : \alpha\}\}\}$$

are equivalent since the latter is the unfolded version of the first type. The following definition of *definitionally equality* ($\equiv$) is capturing this relations.

Definition 3.26 (definitionally equal)**.** Two types $t_1, t_2 \in \mathbb{T}$ are *definitionally equal* ($t_1 \equiv t_2$) iff

- the types are structurally equivalent ($t_1 = t_2$) or
- there is a bijective mapping σ between the sets $G(t_1)$ and $G(t_2)$ such that $\sigma(t_1) = t_2$ or
- one type is the unfolded version of the other, hence $\text{unfold}(t_1) \equiv t_2 \vee t_1 \equiv \text{unfold}(t_2)$ or
- t_1 is
$$rec\ 'i_x.\left\{'i_1 = t_1, \ldots, 'i_k = t_k\right\}$$
and t_2 is
$$rec\ 'j_x.\left\{'j_1 = s_1, \ldots, 'j_l = s_l\right\}$$
and there is a recursive variable instantiation $\sigma_r \in \mathbb{T}_{rvar} \rightharpoonup \mathbb{T}_{rvar}$ such that for every $n \in \{1, \ldots, k\}$ there is a $m \in \{1, \ldots, l\}$ such that $\sigma_r(i_n) = \sigma_r(j_m)$ and $\sigma_r(t_n) \equiv \sigma_r(s_m)$ or
- there is a type t' such that $t_1 \equiv t'$ and $t' \equiv t_2$

The second condition of the definition above ensures that type variables can be consistently renamed without altering the represented value domain. The third identifies recursive types and their unfolded versions. The fourth condition states that the names utilized for recursive type variables are not affecting the represented types – nor is the order in which the nested type bindings are listed. Finally, the last condition ensures transitivity of the definitionally equality relation.

Example 3.5 (definitionally equal)**.** Two generic function types $(\alpha, \alpha) \rightarrow \alpha$ and $(\beta, \beta) \rightarrow \beta$ are definitionally equal since $\sigma = [\alpha \mapsto \beta]$ is bijective and

$$\sigma((\alpha, \alpha) \rightarrow \alpha) = (\beta, \beta) \rightarrow \beta$$

However, neither is definitionally equal to $(\alpha, \beta) \to \beta$ since there is no bijective mapping between the set $G((\alpha, \beta) \to \beta) = \{\alpha, \beta\}$ and the sets $G((\alpha, \alpha) \to \alpha) = \{\alpha\}$ or $G((\beta, \beta) \to \beta) = \{\beta\}$ respectively. Also, the mutual recursive type

$$rec\ \alpha . \{\alpha = struct\,\{a : \beta\}, \beta = struct\,\{b : \alpha\}\}$$

is definitionally equivalent to

$$rec\ \gamma . \{\beta = struct\,\{b : \gamma\}, \gamma = struct\,\{a : \beta\}\}$$

which both are definitionally equivalent to the first types unfolded version

$$struct\,\{a : rec\ \beta . \{\alpha = struct\,\{a : \beta\}, \beta = struct\,\{b : \alpha\}\}\}$$

due to the rule regarding the relation between recursive types and their unfolded form and the transitivity of the definitionally equality relation.

Subtype Relation

Subtype relations represent relations between types $a, b \in \mathbb{T}_c$ where one type is a more restricted version of the other – hence we have $D(a) \subseteq D(b)$. Since the computation of domains is hardly applicable for determining the relation between types the subtype-relation <: is fixed by a set of structural inference rules.

Definition 3.27 (subtypes)**.** The subtype relation is a binary *order relation* (reflexive and antisymmetric) on the set of types $\mathbb{T}$ seeded by the following set of inference rules and may be extended by language extensions regarding their introduced types. The basic set of inference rules include

$$\frac{s \equiv t}{s <: t}\ \text{(ref)}$$

$$\frac{s <: r \qquad r <: t}{s <: t}\ \text{(trans)}$$

$$\frac{\forall i\ .\ \exists j\ .\ n_i = m_j \land s_i \equiv t_j}{union\ \ \{n_1 : s_1, \ldots, n_k : s_k\} <: union\ \ \{m_1 : t_1, \ldots, m_l : t_l\}}\ \text{(union)}$$

$$\frac{\forall i\ .\ t_i <: s_i \qquad s <: t}{(s_1, \ldots, s_k) \to s <: (t_1, \ldots, t_k) \to t}\ \text{(fun)}$$

$$\frac{\forall i\ .\ t_i <: s_i \qquad s <: t}{(s_1, \ldots, s_k) \Rightarrow s <: (t_1, \ldots, t_k) \Rightarrow t}\ \text{(closure)}$$

$$\frac{}{(t_1, \ldots, t_k) \to t <: (t_1, \ldots, t_k) \Rightarrow t}\ \text{(f2c)}$$

Whenever two types are difinitionally equivalent they are sub-types of each other. In particular $t <: t$ for every $t \in \mathbb{T}$. Further, the sub-type relation is transitive, as it is ensured by the (trans) inference rule. The remaining rules fix the implicit sub-type relation between union, function and closure types. In particular every function type $(t_1, \ldots, t_k) \rightarrow t$ is a sub-type of the closure type $(t_1, \ldots, t_k) \Rightarrow t$ as covered by (f2c).

3.5.3 Typing Rules

So far this section was merely handling properties of the type system itself. However, types are assigned to expressions to constrain the set of values they might evaluate to during their evaluation. In the following we will establish an inference system and an algorithm capable of automatically deducing a type for a given IR expression.

Definition 3.28 (typing rules)**.** The typing of expressions is defined by the following set of type inference rules:

$$\frac{e : t \qquad t <: s}{e : s}\ \text{(sub)}$$

$$\frac{}{var(i : t) : t}\ \text{(var)}$$

$$\frac{}{rec(i : t) : t}\ \text{(rec)}$$

$$\frac{}{lit(i : t) : t}\ \text{(lit)}$$

$$\frac{e : (t_1, \ldots, t_k) \Rightarrow t_0 \qquad \forall i\ .\ e_i : \sigma(t_i) \qquad \sigma(t_0) \equiv t}{e(e_1, \ldots, e_k) : t}\ \text{(call)}$$

$$\frac{\forall i\ .\ e_i : t_i}{struct\,\{n_1 = e_1, \ldots, n_k = e_k\} : struct\,\{n_1 : t_1, \ldots, n_k : t_k\}}\ \text{(struct)}$$

$$\frac{e : t}{union\,\{n = e\} : union\,\{n : t\}}\ \text{(union)}$$

$$\frac{e : struct\,\{\ldots, n_x : t, \ldots\}}{e.n_x : t}\ \text{(access struct)}$$

$$\frac{e : union\,\{\ldots, n_x : t, \ldots\}}{e.n_x : t}\ \text{(access union)}$$

$$\frac{\forall i\ .\ v_i : t_i \qquad f : (t_1, \ldots, t_k) \rightarrow t}{rec\ f\,\{\ldots, f = (v_1, \ldots, v_k) \rightarrow t\ s, \ldots\} : (t_1, \ldots, t_k) \rightarrow t}\ \text{(fun)}$$

$$\frac{\forall i \,.\, v_i : t_i \qquad e(e_1, \ldots, e_l) : t}{(v_1, \ldots, v_k) \Rightarrow e(e_1, \ldots, e_l) : (t_1, \ldots, t_k) \Rightarrow t}\ \text{(bind)}$$

$$\frac{}{job\ [e_l, e_u]\ b : job}\ \text{(job)}$$

The first rule (sub) ensures that every expression $e \in \mathbb{E}$ of type $t \in \mathbb{T}$ is also of type $s \in \mathbb{T}$ if s is a super type of t. The remaining inference rules are defined over the structure of IR expressions. For variables and literals the type is already included within the expression. The type of a call expression is inferred from the type of the targeted function, the parameters and a variable instantiation σ integrating the support for generic functions. The types of the five value constructors are inferred recursively from their substructures. Finally, the type of an access construct is determined by the type of the addressed field.

3.5.4 Type Checking and Type Inference

The process of verifying whether a *typing statement* $e : t$ is valid for some expression $e \in \mathbb{E}$ and type $t \in \mathbb{T}$ is known as *type checking*. The typing rules of Definition 3.28 provide the foundation for this procedure. If based on the structure of the typing statement $e : t$ a proof tree can be successfully constructed and all the involved subtype relations are valid, the typing statement is valid.

While the application of most of the inference rules of Definition 3.28 is a mere exercise on matching patterns (var, lit, struct, union, fun, bind, access and job) the rule (call) requires the proper instantiation of the variable substitutions σ. Although this demand does not invalidate its proper definition, indeterministic guessing is something that can hardly be automatized within type verification utilities. We therefore define the process of *type inference* which can be utilized for inferring the required intermediate results.

Type Inference

Every expression may have several associates types. For instance, an expression of type t is also of type s whenever $t <: s$. In general we are interested in determining the most specialized type according to the sub-type relation while selecting the most general generic type in terms of the generality provided by the type variables. We refer to this type as *most general type* (MGT) of an expression. The task of inferring the MGT of an expression is known as *type inference*.

Definition 3.29 (most general type)**.** Let $a, b \in \mathbb{T}$ be types. Type b is more general than type a (denoted by $a \preceq b$) iff

- $b <: a$ or
- $\exists \sigma \,.\, a = \sigma(b)$ or
- $\exists t \in \mathbb{T} \,.\, a \preceq t \wedge t \preceq b$

Furthermore, we will use $a \prec b$ to denote that $a \preceq b$ and $a \neq b$. The *most general type* of an expression $e \in \mathbb{E}$ denoted by MGT(e) is a type $t \in \mathbb{T}$ satisfying

$$e : t \wedge \forall s \in \mathbb{T} \,.\, (e : s \Rightarrow s \preceq t)$$

Hence, any other type that can be validly assigned to expression e is less general than the MGT(e).

Example 3.6 (most general types – generic types)**.** Let f be a generic function of type $(\alpha, \beta) \Rightarrow p\,\langle \alpha, \beta \rangle$, a be an expression of type γ and b be an expression of type δ. Then the expression $f(a, b)$ has the types $p\,\langle \gamma, \delta \rangle$ as well as $p\,\langle \gamma, \gamma \rangle$ since for $\sigma = [\alpha \mapsto \gamma, \beta \mapsto \delta]$ we can prove

$$\frac{f : (\alpha, \beta) \Rightarrow p\,\langle \alpha, \beta \rangle \qquad a : \sigma(\alpha) \qquad b : \sigma(\beta) \qquad \sigma(p\,\langle \alpha, \beta \rangle) \equiv p\,\langle \gamma, \delta \rangle}{f(a, b) : p\,\langle \gamma, \delta \rangle}\ \text{(call)}$$

since $\sigma(\alpha) = \gamma$, $\sigma(\beta) = \delta$ and $\sigma(p\,\langle \alpha, \beta \rangle) = p\,\langle \gamma, \delta \rangle$. Yet utilizing the variable instantiation $\sigma = [\alpha \mapsto \gamma, \beta \mapsto \gamma, \delta \mapsto \gamma]$ we can prove

$$\frac{f : (\alpha, \beta) \Rightarrow p\,\langle \alpha, \beta \rangle \qquad a : \sigma(\alpha) \qquad b : \sigma(\beta) \qquad \sigma(p\,\langle \alpha, \beta \rangle) \equiv p\,\langle \gamma, \gamma \rangle}{f(a, b) : p\,\langle \gamma, \gamma \rangle}\ \text{(call)}$$

where $\sigma(\alpha) = \gamma$, $\sigma(\beta) = \gamma$, $\sigma(p\,\langle \alpha, \beta \rangle) = p\,\langle \gamma, \gamma \rangle$ and $b : \gamma$ is proven by

$$\frac{b : \delta \qquad \dfrac{\delta \equiv \gamma}{\delta <: \gamma}\ \text{(ref)}}{b : \gamma}\ \text{(sub)}$$

However, $p\,\langle \gamma, \delta \rangle \not\equiv p\,\langle \gamma, \gamma \rangle$ since there is a σ such that $\sigma(p\,\langle \gamma, \delta \rangle) = p\,\langle \gamma, \gamma \rangle$ but no σ such that $p\,\langle \gamma, \delta \rangle = \sigma(p\,\langle \gamma, \gamma \rangle)$. However, due to the same reason we have $p\,\langle \gamma, \gamma \rangle \prec p\,\langle \gamma, \delta \rangle$. For the type inference we prefer to assign the more general type $p\,\langle \gamma, \delta \rangle$ to the expression $f(a, b)$ and in fact we have MGT$(f(a, b)) = p\,\langle \gamma, \delta \rangle$.

Example 3.7 (most general types – sub-types)**.** Let $t, s \in \mathbb{T}$ be types such that s is a sub-type of t ($s <: t$). A literal $lit(i : s)$ would then be of type s as well as t since

$$\frac{}{lit(i : s) : s}\ \text{(lit)}$$

as well as

$$\cfrac{\cfrac{}{lit(i:s):s}\,(\text{lit}) \qquad s <: t}{lit(i:s):t}\,(\text{sub})$$

can be proven. However, s is “more general” – although being more specialized in the sub-type hierarchy – according to Definition 3.29 since $s <: t$. Hence, the $\text{MGT}(lit(i:s)) = s$.

Note that for most expression constructs the MGT can be directly derived from its sub-structures or its sub-structures' MGTs. For instance, the most general type of a variable $var(i:t)$ is t and the MGT of a union expression $union\,\{n = e\}$ is $union\,\{n : t\}$ where t is the MGT of the expression e. The only exception is formed by the call expression which is depending on a type-variable instantiation σ. Consequently computing σ is the main challenge when constructing an automated type inference utility.

Definition 3.30 (type inference)**.** Let $s_1, \ldots, s_k \in \mathbb{T}$ be the list of types of the arguments passed to a function call $e(e_1, \ldots, e_k)$ and w.l.o.g.[8] let $(t_1, \ldots, t_l) \to t$ be the type of the targeted function e. If $k \neq l$ the call is not valid and no type can be determined. Otherwise the following procedure computes the most general variable substitution

$$\sigma_{call} = \text{MGS}([s_1, \ldots, s_k], [t_1, \ldots, t_k])$$

such that $\forall_{1 \leq i \leq k} \;.\; s_i <: \sigma_{call}(t_1)$ if such a solution exists. The MGS is required for computing the *most general type*

$$\text{MGT}(e(e_1, \ldots, e_k)) = \sigma_{call}(t)$$

of the processed call expression.

Step 1: Renaming

To avoid invalid type-variable capturing we have to make sure that all type variables within the argument and parameter lists are distinct. Let $\sigma_s, \sigma_t \in \mathbb{T}_{gvar} \rightharpoonup \mathbb{T}_{gvar}$ be two injective[9] variable substitutions mapping type variables to type variables such that

$$\bigcup_{1 \leq i \leq k} G(\sigma_s(s_i)) \cap \bigcup_{1 \leq i \leq k} G(\sigma_t(t_i)) = \emptyset$$

Furthermore let $\sigma_t^{-1} \in \mathbb{T}_{gvar} \rightharpoonup \mathbb{T}_{gvar}$ be the inverse mapping of σ_t such that $\forall v \in \mathbb{T} \;.\; \sigma_t^{-1}(\sigma_t(v)) = v$.

Step 2: Subtype-Dependency Graph Construction

Let $g' \in G = (\mathbb{T}, \mathbb{T} \times \mathbb{T})$ be the directed graph constructed by

$$g' := \bigcup_{1 \leq i \leq k} G_{\preceq}(\sigma_s(s_i), \sigma_t(t_i))$$

[8]Whether the call-target is a function or closure does not effect the inference procedure.
[9]This ensures $\forall i.s_i \equiv \sigma_s(s_i)$ and $\forall i.t_i \equiv \sigma_t(t_i)$.

where

$$G_{\preceq}(s,t) :=$$
$$\begin{cases}
(\emptyset,\emptyset) & \text{if } s = t \text{ or } s <: t \\
(\{s,t\},\{(t,s)\}) & \text{if } s =' i \vee t =' i \\
\bigcup_{1\le i\le k} G_{=}(s_i,t_i) & \text{if } s = i\,\langle s_1,\ldots,s_k\rangle \text{ and } t = i\,\langle t_1,\ldots,t_k\rangle \\
G_{\preceq}(\text{unfold}(s),t) & \text{if } s \text{ is rec-type and } t \text{ is not} \\
G_{\preceq}(s,\text{unfold}(t)) & \text{if } t \text{ is rec-type and } s \text{ is not} \\
G_{\preceq}(s_x,t_y) & \text{if } s = \mathit{rec}\ 'i_x.\{\ldots,' i_x = s_x,\ldots\} \wedge \\
 & \quad t = \mathit{rec}\ 'j_y.\{\ldots,' j_y = t_y,\ldots\} \\
\bigcup_{1\le i\le k} G_{=}(s_i,t_i) & \text{if } s = \mathit{struct}\ \{n_1 : s_1,\ldots,n_k : s_k\} \wedge \\
 & \quad t = \mathit{struct}\ \{n_1 : t_1,\ldots,n_k : t_k\} \\
\bigcup_{1\le i\le l} G_{=}(s_i,t_i) & \text{if } s = \mathit{union}\ \{n_1 : s_1,\ldots,n_l : s_l\} \wedge \\
 & \quad t = \mathit{union}\ \{n_1 : t_1,\ldots,n_k : t_k\} \wedge \\
 & \quad l \le k \\
G_{\preceq}(s_0,t_0) \cup \bigcup_{1\le i\le k} G_{\preceq}(t_i,s_i) & \\
 & \text{if } s = (s_1,\ldots,s_k) \to s_0 \wedge \\
 & \quad t = (t_1,\ldots,t_k) \to t_0 \\
G_{\preceq}(s_0,t_0) \cup \bigcup_{1\le i\le k} G_{\preceq}(t_i,s_i) & \\
 & \text{if } (s = (s_1,\ldots,s_k) \Rightarrow s_0 \vee \\
 & \quad\ s = (s_1,\ldots,s_k) \to s_0) \wedge \\
 & t = (t_1,\ldots,t_k) \Rightarrow t_0 \\
\text{fail} & \text{otherwise}
\end{cases}$$

and

$$G_=(s,t) :=$$

$$\begin{cases}
(\emptyset, \emptyset) & \text{if } s = t \text{ or } s <: t \\
(\{s,t\}, \{(s,t),(t,s)\}) & \text{if } s =' i \vee t =' i \\
\bigcup_{1 \le i \le k} G_=(s_i, t_i) & \text{if } s = i\,\langle s_1, \ldots, s_k\rangle \text{ and } t = i\,\langle t_1, \ldots, t_k\rangle \\
G_=(\text{unfold}(s), t) & \text{if } s \text{ is rec-type and } t \text{ is not} \\
G_=(s, \text{unfold}(t)) & \text{if } t \text{ is rec-type and } s \text{ is not} \\
G_=(s_x, t_y) & \text{if } s = \mathit{rec}\ 'i_x.\{\ldots,' i_x = s_x, \ldots\} \wedge \\
& \quad t = \mathit{rec}\ 'j_y.\{\ldots,' j_y = t_y, \ldots\} \\
\bigcup_{1 \le i \le k} G_=(s_i, t_i) & \text{if } s = \mathit{struct}\ \{n_1 : s_1, \ldots, n_k : s_k\} \wedge \\
& \quad t = \mathit{struct}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\
\bigcup_{1 \le i \le k} G_=(s_i, t_i) & \text{if } s = \mathit{union}\ \{n_1 : s_1, \ldots, n_k : s_k\} \wedge \\
& \quad t = \mathit{union}\ \{n_1 : t_1, \ldots, n_k : t_k\} \\
\bigcup_{0 \le i \le k} G_=(s_i, t_i) & \\
& \text{if } s = (s_1, \ldots, s_k) \rightarrow s_0 \wedge \\
& \quad t = (t_1, \ldots, t_k) \rightarrow t_0 \\
\bigcup_{0 \le i \le k} G_=(s_i, t_i) & \\
& \text{if } (s = (s_1, \ldots, s_k) \Rightarrow s_0 \vee \\
& \quad s = (s_1, \ldots, s_k) \rightarrow s_0) \wedge \\
& t = (t_1, \ldots, t_k) \Rightarrow t_0 \\
\text{fail} & \text{otherwise}
\end{cases}$$

and $\cup : G \times G$ is defined by $(N_1, E_1) \cup (N_2, E_2) = (N_1 \cup N_2, E_1 \cup E_2)$. Let $g' = (N', E')$ and $g \in G$ be the graph obtained by

$$g := (N, E)$$

where N is the closure of N' under the subset relation $<:$ and

$$E := E' \cup \{(t_1, t_2) \in N \mid t_1 <: t_2\}$$

.

Step 3: Compute Strongly Connected Components
Let $c = (N_c, E_c) \in (2^{\mathbb{T}}, 2^{\mathbb{T}} \times 2^{\mathbb{T}})$ be the graph obtained by computing

the graph of *strongly connected components* (SCCs) of graph g. This operation can be conducted using Tarjan's algorithm [96].

Step 4: Unify Strongly Connected Components

All types within the strongly connected components N_c have to be equivalent. We therefore construct a set of unification constraints $a \approx b$ where $a, b \in \mathbb{T}$ by computing

$$u := \bigcup_{c \in N_c} \{t_1 \approx t_2 \mid t_1, t_2 \in c \land t_1 \neq t_2\}$$

and obtain a variable instantiation σ_u by solving those constraints using unification. In case such a solution exists, substitute all mappings $a \mapsto b$ where $a \in \text{dom}(\sigma_s)$ and $b \in \text{dom}(\sigma_t)$ in σ_u by $b \mapsto a$. If there is no solution to the unification problem u the arguments of the processed call are invalid and the type inference process fails.

Step 5: Resolve Constraints

Let $u = (N_u, E_u) \in (\mathbb{T}, \mathbb{T} \times \mathbb{T})$ be the graph obtained from c by replacing each component $c \in N_c$ by a unified represent $\sigma_u(t)$ where $t \in c$. Let $[n_1, \dots, n_k] \in \mathbb{T}^k$ be a reverse-topological ordering of u's nodes. The final variable instantiation σ is computed by

```
$\sigma := \sigma_u$
**for** $i \leftarrow 1 \dots k$ **do**
    $p := \{\sigma(x) \in \mathbb{T} \mid (x, n_i) \in E_u\}$
    **if** $p \neq \emptyset$ **then**
        $t :=$ least_common_super_type($p$)
        **if** $t = \perp$ or $\nexists \tau . \tau(\sigma(n_i)) = t$ **then**
            **return** fail
        **end if**
        $\sigma_t :=$ find $\tau$ such that $\tau(\sigma(n_i)) = t$
        $\sigma := \sigma_t \circ \sigma$
    **end if**
**end for**
```

where the value least_common_super_type(T) is computing a type $r \in \mathbb{T} \cup \{\perp\}$ such that

$$r = \begin{cases} s & \text{if } (\forall t \in T \;.\; t <: s) \land \neg(\exists v \forall t \in T \;.\; t <: v \land v <: s) \\ \perp & \text{if } \neg \exists s \;.\; \forall t \in T \;.\; t <: s \end{cases}$$

and $\sigma_t \circ \sigma$ computes the effective variable instantiation obtained by applying σ_t after σ.

Step 6: Obtain Results

Finally let σ_{call} be $\sigma_t^{-1} \circ \sigma \circ \sigma_t$ and the *most general type* of the processed call expression be $\sigma_{call}(t)$.

Example 3.8 (type inference)**.** Let $A, B, C \in \mathbb{T}$ be three abstract types such that B and C are sub-types of A, hence $B <: A$ and $C <: A$. Further, let f be an expression of the generic type

$$(p\left<\alpha, \beta\right>, (\alpha, \beta) \to \gamma) \to \gamma$$

a be an expression of type

$$p\left<B, C\right>$$

and b be an expression of type

$$(\alpha, \alpha) \to \alpha$$

For instance, f may be a function applying an arbitrary function on the elements of a pair, a may be a pair and b a binary function accepting two arguments of the same type and randomly selecting one of the two. For this example we would like to determine the $\mathrm{MGT}(f(a, b))$ which should be A.

For the type inference procedure of Definition 3.30 we have $s_1 = p\left<B, C\right>$ and $s_2 = (\alpha, \alpha) \to \alpha$ as the argument types and $t_1 = p\left<\alpha, \beta\right>$ and $t_2 = (\alpha, \beta) \to \gamma$ as the parameter types.

Step 1: Renaming

To avoid variable name collisions we utilize the two injective variable mappings

$$\sigma_s = [\alpha \mapsto \alpha_1]$$

and

$$\sigma_t = [\alpha \mapsto \alpha_2, \beta \mapsto \beta_2, \gamma \mapsto \gamma_2]$$

and obtain

$$\sigma_t^{-1} = [\alpha_2 \mapsto \alpha, \beta_2 \mapsto \beta, \gamma_2 \mapsto \gamma]$$

Step 2: Subtype-Dependency Graph Construction

In a first step we construct g' by

$$\begin{aligned} g' &= G_{\preceq}(\sigma_s(s_1), \sigma_t(t_1)) \cup G_{\preceq}(\sigma_s(s_2), \sigma_t(t_2)) \\ &= G_{\preceq}(p\left<B, C\right>, p\left<\alpha_2, \beta_2\right>) \cup G_{\preceq}((\alpha_1, \alpha_1) \to \alpha_1, (\alpha_2, \beta_2) \to \gamma_2) \end{aligned}$$

where

$$\begin{aligned} & G_{\preceq}(p\left<B, C\right>, p\left<\alpha_2, \beta_2\right>) = \\ & \quad = G_{=}(B, \alpha_2) \cup G_{=}(C, \beta_2) \\ & \quad = (\{B, \alpha_2\}, \{(B, \alpha_2), (\alpha_2, B)\}) \cup (\{C, \beta_2\}, \{(C, \beta_2), (\beta_2, C)\}) \\ & \quad = (\{B, \alpha_2, C, \beta_2\}, \{(B, \alpha_2), (\alpha_2, B), (C, \beta_2), (\beta_2, C)\}) \end{aligned}$$

and

$$\begin{aligned}
&G_{\preceq}((\alpha_1,\alpha_1) \to \alpha_1, (\alpha_2,\beta_2) \to \gamma_2) = \\
&\qquad = G_{\preceq}(\alpha_1,\gamma_2) \cup G_{\preceq}(\alpha_2,\alpha_1) \cup G_{\preceq}(\beta_2,\alpha_1) \\
&\qquad = (\{\alpha_1,\alpha_2,\beta_2,\gamma_2\}, \{(\gamma_2,\alpha_1),(\alpha_1,\alpha_2),(\alpha_1,\beta_2)\})
\end{aligned}$$

so that we obtain

$$\begin{aligned}
&g' = (\\
&\quad \{B,\alpha_2,C,\beta_2,\alpha_1,\gamma_2\}, \\
&\quad \{(B,\alpha_2),(\alpha_2,B),(C,\beta_2),(\beta_2,C),(\gamma_2,\alpha_1),(\alpha_1,\alpha_2),(\alpha_1,\beta_2)\} \\
&)
\end{aligned}$$

which is illustrated by

$$\begin{tikzcd}
 & & \alpha_2 \arrow[r, bend left] & B \arrow[l, bend left] \\
\gamma_2 \arrow[r] & \alpha_1 \arrow[ur] \arrow[dr] & & \\
 & & \beta_2 \arrow[r, bend left] & C \arrow[l, bend left]
\end{tikzcd}$$

To obtain the full subtype-dependency graph g we adding all the super-types of the present types and corresponding edges. In our case we have to add the type A and edges from B and C to A so that we obtain the graph g:

$$\begin{tikzcd}
 & & \alpha_2 \arrow[r, bend left] & B \arrow[l, bend left] \arrow[dr] & \\
\gamma_2 \arrow[r] & \alpha_1 \arrow[ur] \arrow[dr] & & & A \\
 & & \beta_2 \arrow[r, bend left] & C \arrow[l, bend left] \arrow[ur] &
\end{tikzcd}$$

Step 3: Computing Strongly Connected Components

For this step the graph c of SCCs of graph g is computed. The result corresponds to:

$$\begin{tikzcd}
 & & \{\alpha_2,B\} \arrow[dr] & \\
\{\gamma_2\} \arrow[r] & \{\alpha_1\} \arrow[ur] \arrow[dr] & & \{A\} \\
 & & \{\beta_2,C\} \arrow[ur] &
\end{tikzcd}$$

Step 4: Unify Strongly Connected Components

From graph c we obtain the unification constraints

$$u = \{\alpha_2 \approx B, \beta_2 \approx C\}$$

from which we obtain the variable instantiation

$$\sigma_u = \epsilon[\alpha_2 \mapsto B][\beta_2 \mapsto C]$$

Step 5: Resolve Constraints

Based on σ_u we obtain graph u by replacing the nodes of c by their unified represent:

$$\begin{tikzcd}
 & & B \arrow[dr] & \\
\gamma_2 \arrow[r] & \alpha_1 \arrow[ur] \arrow[dr] & & A \\
 & & C \arrow[ur] &
\end{tikzcd}$$

A reverse-topological order of u's nodes is given by

$$[n_1, \ldots, n_5] = [A, B, C, \alpha_1, \gamma_2]$$

In the next step we obtain σ by processing the algorithm of Step 5 of Definition 3.30 as follows:

iteration	σ before	n_i	p	t	σ_t	σ after
$\langle init \rangle$	σ_u	–	–	–	–	σ_u
1	σ_u	A	$\emptyset$	–	–	σ_u
2	σ_u	B	$\emptyset$	–	–	σ_u
3	σ_u	C	$\emptyset$	–	–	σ_u
4	σ_u	α_1	$\{B, C\}$	A	$\epsilon[\alpha_1 \mapsto A]$	$\sigma_u[\alpha_1 \mapsto A]$
5	$\sigma_u[\alpha_1 \mapsto A]$	γ_2	$\{A\}$	A	$\epsilon[\gamma_2 \mapsto A]$	$\sigma_u[\alpha_1 \mapsto A][\gamma_2 \mapsto A]$

Resulting in

$$\sigma = [\alpha_2 \mapsto B, \beta_2 \mapsto C, \alpha_1 \mapsto A, \gamma_2 \mapsto A]$$

Step 6: Obtain Results

Finally we obtain

$$\begin{aligned}\sigma_{call} &= \sigma_t^{-1} \circ \sigma \circ \sigma_t \\ &= [\alpha \mapsto B, \beta \mapsto C, \gamma \mapsto A, \alpha_2 \mapsto B, \beta_2 \mapsto C, \alpha_1 \mapsto A, \gamma_2 \mapsto A]\end{aligned}$$

When applied to the result type γ of the input function f we obtain

$$\sigma_{call}(\gamma) = A$$

and hence $\mathrm{MGT}(f(a, b)) = A$

The procedure of Definition 3.30 therefore correctly obtains $A \in \mathbb{T}$ as the most general type of the call expression $f(a, b)$.

Type Checking Procedure

Based on the type inference procedure for *call expressions* presented above an automated type checking utility can be realized based on the recursive, bottom-up computation of the *most general types* of expressions.

Definition 3.31 (type checking procedure)**.** Let $e \in \mathbb{E}$ be an expression and $MGT(e) \in \mathbb{T}$ be its most general type computed by

$$\mathrm{MGT}(e) := \begin{cases} t & \text{if } e = var(i : t) \\ t & \text{if } e = lit(i : t) \\ \sigma_{call}(t) & \text{if } e = e_0(e_1, \ldots, e_n) \text{ and } \sigma_{call} \text{ from Definition 3.30} \\ struct\,\{n_1 : \mathrm{MGT}(e_1), \ldots, n_k : \mathrm{MGT}(e_k)\} & \\ & \text{if } e = struct\,\{n_1 = e_1, \ldots, n_k = e_k\} \\ union\,\{n : \mathrm{MGT}(e_m)\} & \\ & \text{if } e = union\,\{n = e_m\} \\ t & \text{if } e = e_0.n \text{ and } \mathrm{MGT}(e_0) = struct\,\{\ldots, n : t, \ldots\} \text{ or} \\ & \qquad\qquad\qquad \mathrm{MGT}(e_0) = union\,\{\ldots, n : t, \ldots\} \\ (\mathrm{MGT}(v_1), \ldots, \mathrm{MGT}(v_k)) \to t & \\ & \text{if } e = rec\ f_x.\{\ldots, f_x = (v_1, \ldots, v_k) \to t\ s \ldots, \ldots\} \\ t & \text{if } e = rec(i : t) \\ (\mathrm{MGT}(v_1), \ldots, \mathrm{MGT}(v_k)) \Rightarrow \mathrm{MGT}(e(e_1, \ldots, e_l)) & \\ & \text{if } e = (v_1, \ldots, v_k) \Rightarrow e(e_1, \ldots, e_l) \\ job & \text{if } e = job\ [e_l, e_u]\ b \end{cases}$$

A *typing statement* $e : t$ is valid iff $\mathrm{MGT}(e) <: t$.

3.6 Valid Code Fragments

Unfortunately not every expression in $\mathbb{E}$ and statement in $\mathbb{S}$ is a *valid* representation of a program or code fragment. For instance, functions might be called by passing an invalid number of arguments, variables might be used without being defined before or *return* statements might return values of a type not matching the result type of the enclosing function. Constructing an abstract language syntax enforcing this kind of constraints would result in overly complex constructs. Therefore, for the design of our IR, we have followed the common approach of specifying a simple syntax (see Section 3.4) followed by a set of restrictions which have to be satisfied to form a *valid code fragment.* Within this section those restrictions are covered.

The restrictions are sub-divided into two categories – those imposed on expressions and those imposed on statements. Both categories, *valid expressions* ($\mathbb{E}_v$) and *valid statements* ($\mathbb{S}_v$) are covered within the following sub-sections. To simplify the definition of the constraints we require some auxiliary definitions.

3.6.1 Auxiliary Definitions

In a first step we extend the definition of free type variables $F(t)$ (see Definition 3.16) to cover expressions and statements.

Definition 3.32 (free type variables for statements)**.** Let $s \in \mathbb{S}$ be a statement. The set $F(s) \subset \mathbb{T}_{gvar}$ of free type variables is defined by

$$F(s) := \begin{cases} F(MGT(e)) & \text{if } s = e \in \mathbb{E} \\ \bigcup_{1 \leq i \leq n} F(s_i) & \text{if } s = \{s_1; \ldots; s_n\} \\ F(e) \cup F(s_1) \cup F(s_2) & \text{if } s = if \ \ (e) \ \ then \ s_1 \ else \ s_2 \\ F(e) \cup F(b) & \text{if } s = while \ \ (e) \ \ do \ b \\ F(v) \cup \bigcup_{i=1}^{3} F(e_i) \cup F(b) & \text{if } s = for \ \ (v = e_1 \ldots e_2 : e_3) \ \ do \ b \\ F(e) & \text{if } s = return \ e \\ \emptyset & \text{otherwise} \end{cases}$$

Also, we define a function $F_{var} : \mathbb{S} \to 2^{\mathbb{V}}$ computing the set of free variables within expressions and statements.

Definition 3.33 (free variables)**.** Let $e \in \mathbb{E}$ be an expression. The set $F'_{var}(e) \subset \mathbb{V}$ of *free variables* within expression e is defined by

$$F'_{var}(e) := \begin{cases} \{e\} & \text{if } e = var(i : t) \\ \emptyset & \text{if } e = lit(i : t) \\ \bigcup_{i=0}^{k} F_{var}(e_i) & \text{if } e = e_0(e_1, \ldots, e_k) \\ \bigcup_{i=1}^{k} F_{var}(e_i) & \text{if } e = struct\,\{n_1 = e_1, \ldots, n_k = e_k\} \\ F_{var}(e_1) & \text{if } e = union\,\{n = e_1\} \\ F_{var}(e_1) & \text{if } e = e_1.n \\ \emptyset & \text{if } e = rec \ f_x.\{\ldots\} \\ \emptyset & \text{if } e = rec(i : t) \\ \bigcup_{i=0}^{l} F_{var}(e_i) \setminus \{v_1, \ldots, v_k\} & \\ & \text{if } e = (v_1, \ldots, v_k) \Rightarrow e_0(e_1, \ldots, e_l) \\ F_{var}(e_l) \cup F_{var}(e_u) \cup F_{var}(b) & \\ & \text{if } e = job \ [e_l, e_u] \ b \end{cases}$$

Let $s \in \mathbb{S}$ be a statement. The set $F_{var}(s) \subset \mathbb{V}$ of *free variables* within statement s is defined by

$$F_{var}(s) := \begin{cases} F'_{var}(s) & \text{if } s \in \mathbb{E} \\ \emptyset & \text{if } s = \{\} \\ F_{var}(s_1) \cup F_{var}(\{s_2, \ldots, s_n\}) & \\ \quad & \text{if } s = \{s_1, \ldots, s_n\} \wedge n > 0 \wedge s_1 \neq decl\ v = e \\ F_{var}(e) \cup (F_{var}(\{s_2, \ldots, s_n\}) \setminus \{v\}) & \\ \quad & \text{if } s = \{s_1, \ldots, s_n\} \wedge n > 0 \wedge s_1 = decl\ v = e \\ F_{var}(e) \cup F_{var}(s_1) \cup F_{var}(s_2) & \\ \quad & \text{if } s = if\ \ (e)\ \ then\ s_1\ else\ s_2 \\ F_{var}(e) \cup F_{var}(b) & \\ \quad & \text{if } s = while\ \ (e)\ \ do\ b \\ \bigcup_{i=1}^{3} F_{var}(e_i) \cup (F_{var}(b) \setminus \{v\}) & \\ \quad & \text{if } s = for\ \ (v = e_1 \ldots e_2 : e_3)\ \ do\ b \\ F_{var}(e) & \text{if } s = return\ e \\ \emptyset & \text{if } s = break \\ \emptyset & \text{if } s = continue \end{cases}$$

Similar we define a function $F_{rec} : \mathbb{S} \to 2^{\mathbb{V}_{rec}}$ computing the set of free recursive variables within expressions and statements.

Definition 3.34 (free recursive variables)**.** Let $e \in \mathbb{E}$ be an expression. The set $F'_{rec}(e) \subset \mathbb{V}_{rec}$ of *free recursive variables* within expression e is defined by

$$F'_{rec}(e) := \begin{cases} \emptyset & \text{if } e = v \\ \emptyset & \text{if } e = lit(i : t) \\ \bigcup_{i=0}^{k} F_{rec}(e_i) & \text{if } e = e_0(e_1, \ldots, e_k) \\ \bigcup_{i=1}^{k} F_{rec}(e_i) & \text{if } e = struct\,\{n_1 = e_1, \ldots, n_k = e_k\} \\ F_{rec}(e_1) & \text{if } e = union\,\{n = e_1\} \\ F_{rec}(e_1) & \text{if } e = e_1.n \\ \bigcup_{i=1}^{k} F_{rec}(s_i) \setminus \{f_1, \ldots, f_k\} & \\ \quad & \text{if } e = rec\ f_x.\{ \\ & \qquad f_1 = (v_{11}, \ldots, v_{1l_1}) \to t_1\ s_1, \\ & \qquad \ldots, \\ & \qquad f_k = (v_{k1}, \ldots, v_{kl_k}) \to t_k\ s_k \\ & \quad \} \\ \{e\} & \text{if } e = rec(i : t) \\ \bigcup_{i=0}^{l} F_{rec}(e_i) & \text{if } e = (v_1, \ldots, v_k) \Rightarrow e_0(e_1, \ldots, e_l) \\ F_{rec}(e_l) \cup F_{rec}(e_u) \cup F_{rec}(b) & \\ \quad & \text{if } e = job\ [e_l, e_u]\ b \end{cases}$$

Let $s \in \mathbb{S}$ be a statement. The set $F_{rec}(s) \subset \mathbb{V}_{rec}$ of *free recursive variables* within statement s is defined by

$$F_{rec}(s) := \begin{cases} F'_{rec}(s) & \text{if } s \in \mathbb{E} \\ \bigcup_{i=1}^{n} F_{rec}(s_i) & \text{if } s = \{s_1, \dots, s_n\} \\ F_{rec}(e) \cup F_{rec}(s_1) \cup F_{rec}(s_2) & \\ & \text{if } s = if\ \ (e)\ \ then\ s_1\ else\ s_2 \\ F_{rec}(e) \cup F_{rec}(b) & \\ & \text{if } s = while\ \ (e)\ \ do\ b \\ \bigcup_{i=1}^{3} F_{rec}(e_i) \cup F_{rec}(b) & \\ & \text{if } s = for\ \ (v = e_1 \dots e_2 : e_3)\ \ do\ b \\ F_{rec}(e) & \text{if } s = return\ e \\ \emptyset & \text{if } s = break \\ \emptyset & \text{if } s = continue \end{cases}$$

Furthermore, the scope of *break* and *return statements* has to be fixed. As for other constructs we do so by providing a definition for free break and return statements.

Definition 3.35 (free break and return statements)**.** Let $s \in \mathbb{S}$ be a statement. The set $F_{brk}(s)$ of *free break statements* is defined by

$$F_{brk}(s) := \begin{cases} \bigcup_{i=1}^{n} F_{brk}(s_i) & \text{if } s = \{s_1; \dots; s_n\} \\ F_{brk}(s_1) \cup F_{brk}(s_2) & \text{if } s = if\ \ (e)\ \ then\ s_1\ else\ s_2 \\ \{break\} & \text{if } s = break \\ \emptyset & \text{otherwise} \end{cases}$$

and the set $F_{ret}(s)$ of *free return statements* is defined by

$$F_{ret}(s) := \begin{cases} \bigcup_{i=1}^{n} F_{ret}(s_i) & \text{if } s = \{s_1; \dots; s_n\} \\ F_{ret}(s_1) \cup F_{ret}(s_2) & \text{if } s = if\ \ (e)\ \ then\ s_1\ else\ s_2 \\ F_{ret}(b) & \text{if } s = while\ \ (e)\ \ do\ b \\ F_{ret}(b) & \text{if } s = for\ \ (v = e_s \dots e_e : e_i)\ \ do\ b \\ \{s\} & \text{if } s = return\ e \\ \emptyset & \text{otherwise} \end{cases}$$

Note that $F_{brk}(s) \in \{\emptyset, \{break\}\}$ for any statement s while $F_{ret}(s)$ is collecting all return statements including the expressions computing the value to be returned. Also, the scopes of *break* statements are bound by function bodies and loops while *return* statements are only bound by function bodies.

3.6.2 Valid Expressions

Many of the restrictions of *valid expressions* are imposed on the function construct. We therefore provide a separated definition for valid functions.

Definition 3.36 (valid functions)**.** A function expression

$$\begin{aligned} e = rec\ f_x.\{ \\ & f_1 = (var(i_{11} : t_{11}), \ldots, var(i_{1l_1} : t_{1l_1})) \rightarrow t_1\ s_1, \\ & \ldots, \\ & f_k = (var(i_{k1} : t_{k1}), \ldots, var(i_{kl_k} : t_{kl_k})) \rightarrow t_k\ s_k \\ \} \end{aligned}$$

of type $t_e = MGT(e)$ is a *valid function* iff the following constraints are satisfied:

- for all $1 \leq i \leq k$ the type of f_i is $(t_{i1}, \ldots, t_{il_i}) \rightarrow t_i$
- for all $1 \leq i \leq k$ the statement s_i is a valid statement ($s_i \in \mathbb{S}_v$)
- for all $1 \leq j \leq k$ we have

  $$F_{var}(s_j) \subseteq \{var(i_{j1} : t_{j1}), \ldots, var(i_{jl_j} : t_{jl_j})\}$$

  Hence the function body s_j has no free variables except for the associated parameters.
- for all $1 \leq i \leq k$ we have $(F_{ret}(s_i) = \emptyset) \Rightarrow (t_i = unit)$ and $\forall return\ e \in F_{ret}(s_i)\ .\ e : t_i$ – hence, if there are no free returns, the return type of the function is *unit*; otherwise the returned values have to be consistent with the return type
- for all $1 \leq i \leq k$ we have $F(s_i) \subseteq G(MGT(f_i))$ – hence all free type variables within the bodies are generic type variables of the associated function type

Furthermore, e is said to be a *valid generic function* if $G(t_e) \neq \emptyset$.

Based on the definition of valid functions we can extend the definition of validity to all expressions as follows:

Definition 3.37 (valid expressions)**.** An expression $e \in \mathbb{E}$ is valid iff

- it can be typed, hence its most general type $t = MGT(e)$ can be computed and $t \in \mathbb{T}_v$ is a *valid type*
- if e is a literal representing a constant or an abstract operator it has to be one of the built-in literals of the language core or defined by some known extension (see Section 3.8)
- if e is a function expression, it is a *valid function expression*
- if e is a bind expression $(v_1, \ldots, v_k) \Rightarrow e_0(e_1, \ldots, e_l)$ then for all $0 \leq i \leq l$ we have $e_i \in \{v_1, \ldots, v_k\}$ or $F_{var}(e_i) \cap \{v_1, \ldots, v_k\}$ is empty.

- if e is a job expression $job\ [e_l, e_u]\ b$ then $MGT(e_l) = MGT(e_u) \prec uint\langle\alpha\rangle$[10] and $b : () \Rightarrow unit$ has to be valid
- all sub-expressions of e are valid expressions too

The set of all *valid expressions* satisfying these constraints is denoted by $\mathbb{E}_v \subset \mathbb{E}$.

3.6.3 Valid Statements

Also the construction of statements is constraint by the composition and types of its sub-components.

Definition 3.38 (valid statements)**.** A statement $s \in \mathbb{S}$ is valid iff all sub-expressions and statements are valid statements and one of the following conditions is satisfied:

- if s is an expression it is a valid expression ($s \in \mathbb{E}_v$)
- if s is a *declaration statement* of the shape $decl\ var(i : t) = e$ then $e : t$ has to be valid
- if s is a *conditional statement* of the shape $if\ \ (e)\ \ then\ s_1\ else\ s_2$ then $e : bool$ has to be valid[11].
- if s is a *while statement* of the shape $while\ \ (e)\ \ do\ b$ then $e : bool$ has to be valid
- if s is a *for statement* of the shape $for\ \ (v = e_s \ldots e_e : e_i)\ \ do\ b$ then there has to be a type $t \in \mathbb{T}_c$ such that $t = MGT(v)$, $e_s : t$, $e_e : t$, and $e_i : t$ hold and $t \prec int\langle\alpha\rangle$ or $t \prec uint\langle\alpha\rangle$. Furthermore, b must not contain any free *break* or *return statements* ($F_{brk}(b) = F_{ret}(b) = \emptyset$).

Analogous to the expressions, the set of all *valid statements* is denoted by $\mathbb{S}_v \subset \mathbb{S}$.

3.6.4 Valid Programs

In a final step we are able to provide a definition for a valid IR program.

Definition 3.39 (valid program)**.** The set $\mathbb{P}$ of valid INSPIRE programs is defined by

$$\mathbb{P} = \{e \in \mathbb{E}_v \mid F(e) = F_{var}(e) = F_{rec}(e) = \emptyset\}$$

Hence, every valid expression e not exhibiting any free type variables, free variables or free recursive variables is a valid INSPIRE program.

[10]The abstract parametrized types $int\langle\alpha\rangle$ and $uint\langle\alpha\rangle$ model signed and unsigned integral numbers – see Section 3.8.2.

[11]The abstract type $bool$ models the truth values $true$ and $false$ – see Section 3.8.2.

By definition, a valid program must not exhibit any free type variables. Nevertheless, it may contain properly nested generic functions and non-closed sub-expressions and statements.

3.7 Semantic

After INSPIRE's syntax has been introduced within Section 3.4 and validity restrictions have been covered within Section 3.6 this section is formalizing the language's semantic.

3.7.1 The Small-Step Transition Relation

To formalize the semantic of INSPIRE's language constructs, its operational semantic will be elaborated within this section. The general idea is to define a relation restricting valid transitions between *global program states*. Based on an initial state constructed from a valid INSPIRE program, the set of potential program traces is fixed by the transitive closure of the *global program state transition relation* to be covered in this section.

INSPIRE supports the incorporation of abstract functions and symbols. Consequently, to provide a semantic interpretation of an INSPIRE code fragment, interpretations of those abstract elements have to be provided. This section will start by specifying how those interpretations are integrated. It is followed by the specification of the *program states* which are then utilized for formalizing the desired state transitions.

Abstract Symbol Interpretations

In INSPIRE, abstract constants and functions can be incorporated within code fragments utilizing *literal* expressions. To determine the semantic of code fragments including abstract elements, interpretations for those are required. Those interpretations have to be provided by the corresponding language extensions introducing the abstract constructs. Formally, an interpretation is a mapping from *literals* to *values* of a universal *value set.*

Definition 3.40 (universal value set)**.** The set of all concrete values processed by an INSPIRE program is denoted by $\mathcal{V}$.

We leave the exact definition of $\mathcal{V}$ open such that it can be extended by extensions as required. Any extension may add additional objects to be included within $\mathcal{V}$. For instance, the arithmetic extension introduces $\mathbb{Z}$ as a subset of $\mathcal{V}$ since integer literals are representing corresponding values. Also the interpretation of arithmetic operations like addition ($+$) and multiplication ($*$) are included in $\mathcal{V}$ since literals representing these operations are covered by the corresponding extensions.

The interpretation of abstract literals is a mapping from IR literals to the universal value set $\mathcal{V}$.

Definition 3.41 (interpretation)**.** The set of all *interpretations* $\mathcal{I}$ is defined by the set of partial mappings

$$\mathcal{I} = (\{lit(i : t) \in \mathbb{E}_v\} \rightharpoonup \mathcal{V})$$

where $\{lit(i : t) \in \mathbb{E}_v\}$ is the set of all valid literals in INSPIRE and $\mathcal{V}$ the universal value set.

To define the semantic of a program code, the interpretation of all its abstract symbols is required. The complete interpretation is obtained by concatenating the interpretations provided by the individual language extensions referenced by an IR code fragment. For the remainder of this section we assume that $I \in \mathcal{I}$ is the aggregation of all those interpretations.

Program States

The state of a system is modeled by a pair of elements. The first element covers the state of the system the program is manipulating while the second component models the progress of the processed program. To model the former we utilize the following environment definition.

Definition 3.42 (environment)**.** An *environment* is a mapping between an arbitrary key set $\mathcal{K}$ and the value set $\mathcal{V}$. The set of all environments $\mathcal{E}$ is given by

$$\mathcal{E} = (\mathcal{K} \rightharpoonup \mathcal{V})$$

Note that we have left the set of keys $\mathcal{K}$ undefined within the definition of environments the same way as we kept the set of values $\mathcal{V}$ open for extensions. On the one hand, a restriction of those two sets is not required for the following definitions and on the other hand this leaves the environment open to be utilized for modeling the effect of operations introduced by language extensions.

Frequently the content to be maintained within environments are (partial) functions. This can be quite cumbersome. For instance, to realize an event counter for a set of events E within an environment e we would have to define a grammar for keys, e.g. $k ::= counter(i)$ where $i \in E$ is an event identifier, which is later on utilized for read and update operations. In this example, $e[counter(i)]$ is obtaining the current counter value of event i and the term

$$e[counter(i) \mapsto e[counter(i)] + 1]$$

is an example of an environment where the value of the counter has been increased by 1.

To mitigate the need of extra grammars and excessive terms regarding update operations of functions we introduce an abbreviated notation for functions stored within environments.

Definition 3.43 (environment function notation)**.** Let f be the name of a (partial) n-ary function $(A_1 \times \ldots \times A_n) \rightarrow B$ to be maintained within an environment $e \in \mathcal{E}$ for arbitrary sets $A_1, \ldots, A_n$, and B. Let $f_\perp \in B$ be the default value to be assigned to values not within the domain of f ($f_\perp$ may be left *undefined*). Then the value $f_e(a_1, \ldots, a_n)$ is defined by

$$f_e(a_1, \ldots, a_n) := \begin{cases} e[\text{'f'}(a_1, \ldots, a_n)] & \text{if 'f'}(a_1, \ldots, a_n) \in \text{dom}(e) \\ f_\perp & \text{otherwise} \end{cases}$$

where 'f'$(a_1, \ldots, a_n)$ is an element of the induced key-grammar

$$k ::= \text{'f'}(e_1, \ldots, e_n)$$

where $e_i \in A_i$ for all $1 \leq i \leq n$. Furthermore, the expression

$$f_e(a_1, \ldots, a_n) \leftarrow b$$

is abbreviating

$$e[\text{'f'}(a_1, \ldots, a_n) \mapsto b]$$

hence the update of the function value $f(a_1, \ldots, a_n)$ to match b within environment e. Finally, let $g : (C_1 \times \ldots \times C_i \times B \times C_{i+1} \times \ldots C_k) \rightarrow B$ be a function over the co-domain of function f and some extra parameters of some arbitrary sets $C_1, \ldots, C_k$. Then the term

$$g(c_1, \ldots, c_i, f_e(a_1, \ldots, a_n), c_{i+1}, \ldots, c_k)$$

is considered equivalent to

$$f_e(a_1, \ldots, a_n) \leftarrow g(c_1, \ldots, c_i, f_e(a_1, \ldots, a_n), c_{i+1}, \ldots, c_k)$$

when utilized in a context demanding an environment.

Example 3.9 (counter)**.** Based on the environment function notation the counter from the motivating example above can be realizes as follows: Let $\mathit{counter} : E \rightarrow \mathbb{N}$ be a (partial) function for a set of events E with the default value $\mathit{counter}_\perp = 0$. To read the current value of the counter within environment $e \in \mathcal{E}$ for an event $i \in E$ we can use $\mathit{counter}_e(i)$. Note that if there is no corresponding value covered by e the result will be 0. To obtain an environment where the value of the counter of event i has been incremented by 1 we can use the term $\mathit{counter}_e(i){+}{+}$ by utilizing the familiar post-fix notation of the *increment operation* $++ : \mathbb{N} \rightarrow \mathbb{N}$ defined by $x = x + 1$.

Intermediate Statements The second component of the program state is modeling the program progress itself. The progress is given by the remaining code fragment to be processed. However, for formalizing the effects of various language constructs the static IR syntax has to be extended by some additional intermediate elements. The following definition introduces the sets of *intermediate variables* $\mathbb{IV}$, *intermediate expressions* $\mathbb{IE}$ and *intermediate statements* $\mathbb{IS}$.

Definition 3.44 (intermediate expressions and statements)**.** The set *intermediate variables* $\mathbb{IV}$ is defined by

$$\mathbb{IV} = \{v : t \mid v \in \mathbb{V}\}$$

Let $\nu_* \in \mathcal{V}$ be values, $i \in \mathbb{I}$ be an identifier, $t_* \in \mathbb{T}$ be types, $n_* \in \mathbb{I}$ be identifiers utilized as names, $e_*, b \in \mathbb{IE}$ be intermediate expressions, $v_* \in \mathbb{IV}$ be intermediate variables, $f_* \in \mathbb{V}_{rec}$ be recursive variables and $s_* \in \mathbb{IS}$ be intermediate statements. Then, for all $x > 0 \in \mathbb{N}$,

$$\begin{aligned}
&v && \text{(var)}\\
&lit(i:t):t && \text{(lit)}\\
&e(e_1,\dots,e_k):t && \text{(call)}\\
&struct\,\{n_1 = e_1,\dots,n_k = e_k\}:t && \text{(struct)}\\
&union\,\{n = e\}:t && \text{(union)}\\
&e.n:t && \text{(access)}\\
&rec\ f_x.\{\\
&\qquad f_1 = (v_{11},\dots,v_{1l_1}) \to t_1\ s_1,\\
&\qquad \dots,\\
&\qquad f_k = (v_{k1},\dots,v_{kl_k}) \to t_k\ s_k\\
&\}:t && \text{(func)}\\
&f:t && \text{(rec)}\\
&(v_1,\dots,v_k) \Rightarrow e(e_1,\dots,e_l):t && \text{(bind)}\\
&job\ [e_l,e_u]\ b:t && \text{(job)}\\
&\nu:t && \text{(value)}\\
&(v_1,\dots,v_k) \to s:t && \text{(callable)}\\
&eval(s):t && \text{(eval)}\\
&[\nu_l,\nu_u] \to b:t && \text{(spawnable)}
\end{aligned}$$

are intermediate expressions as well. The set $\mathbb{IE}$ is the smallest set being closed under those constructs.

Let $e_* \in \mathbb{IE}$ be intermediate expressions, $v \in \mathbb{IV}$ be an intermediate variable, $t \in \mathbb{T}$ be a type and $s_* \in \mathbb{IS}$ be intermediate statements. Then, for all $n \in \mathbb{N}$

$$
\begin{aligned}
&e && \text{(expr)}\\
&\mathit{decl}\ v = e && \text{(decl)}\\
&\{s_1; \ldots; s_n\} && \text{(comp)}\\
&\mathit{if}\ \ (e)\ \ \mathit{then}\ s_1\ \mathit{else}\ s_2 && \text{(if)}\\
&\mathit{while}\ \ (e)\ \ \mathit{do}\ s && \text{(while)}\\
&\mathit{for}\ \ (v = e_s \ldots e_e : e_i)\ \ \mathit{do}\ s && \text{(for)}\\
&\mathit{return}\ e && \text{(return)}\\
&\mathit{break} && \text{(break)}\\
&\mathit{continue} && \text{(continue)}\\
&\mathit{loop}\ (s) && \text{(loop)}
\end{aligned}
$$

are intermediate statements as well. The set $\mathbb{IS}$ is the smallest set being closed under those constructs.

Note that the definition of intermediate expressions is analogous to the definition of ordinary IR expressions with the distinction that every construct is annotated by a type and a few extra constructs have been introduced. The additional constructs ((value), (callable), (eval), (spawnable), and (loop)) are required for formalizing the effects of the remaining constructs.

We apply the same syntactic sugar conventions to intermediate expressions and statements as we do for ordinary IR constructs. Additionally we may omit the trailing type $: t$ if its presence is clear from the context. In particular, a term $\nu : t \in \mathbb{IS}$ and its value counterpart $\nu \in \mathcal{V}$ are considered interchangeable if their relation is clear from the context.

Intermediate statements are utilized to represent programs within the state transitions developed within this section. Consequently, since the objective is to specify the semantic of INSPIRE constructs a conversion between ordinary IR statements and intermediate IR statements is required.

Definition 3.45 (type annotations)**.** The function conv : $\mathbb{S} \to \mathbb{IS}$ replaces every expression $e \in \mathbb{E}$ by $e : \mathit{MGT}(e) \in \mathbb{IS}$ within a given IR statement. Further, the function type : $\mathbb{IE} \to \mathbb{T}$ is defined by $\_ : t \mapsto t$ where $\_$ is a wild-card for any of the intermediate expression constructors.

Essentially the function conv is annotating every expression within an IR code fragment with its most general type and the type function is extracting this annotated type from an intermediate expression.

Irreducible Expressions The basic idea of the following transition relation is to model the gradual reduction of a program (=expression) from an initial state to a resulting value. Thus a definition of a value distinguishing it from something that can be further processed is required. For this purpose the set of *irreducible expressions* is introduced.

Definition 3.46 (irreducible expression)**.** An expression $e \in \mathbb{IE}$ is *irreducible* if one of the following constraints is satisfied:

- $e = \nu : t$ and $\nu \in \mathcal{V}$ – hence e is a value or
- $e = lit(i : t) : t$ and $e \notin \text{dom}(I)$ or
- $e = (v_1, \ldots, v_k) \rightarrow s : t$ where $v_1, \ldots, v_k \in \mathbb{IV}$ and $s \in \mathbb{IS}$ or
- $e = [\nu_l, \nu_u] \rightarrow b : t$ where $\nu_l, \nu_u \in \mathcal{V}$ and $b \in \mathbb{IS}$

The set of irreducible expressions is denoted by $\mathbb{IE}_i \subset \mathbb{IE}$.

For the remainder of this section placeholders of the shape ν_* are considered to be restricted to *irreducible expressions* unless explicitly stated otherwise.

Synchronizing Expressions For the specification of the IR language constructs we have to further distinguish between operations whose effects are isolated to a single thread and operations with *global synchronizing side effects*. The latter group is defined by the set of *global synchronizing expressions*.

Definition 3.47 (global synchronizing expressions)**.** An intermediate expression $e \in \mathbb{IE}$ is a *global synchronizing expression* iff e equals

- $spawn(\nu) : t$ for some $\nu \in \mathbb{IE}_i$ and $t \in \mathbb{T}$ or
- $merge(\nu) : t$ for some $\nu \in \mathbb{IE}_i$ and $t \in \mathbb{T}$ or
- $merge() : t$ for some $t \in \mathbb{T}$ or
- $redistribute(\nu_1, \nu_2) : t$ for some $\nu_1, \nu_2 \in \mathbb{IE}_i$ and $t \in \mathbb{T}$

where *spawn*, *merge*, and *redistribute* are the literals introduced within section 3.4.2 converted into intermediate expressions. The set of *global synchronizing expressions* is denoted by $\mathbb{IE}_{gsync}$

Thread Addresses Another required component for modeling program states is a way to address threads. Threads are organized in a hierarchy of thread groups. To identify a thread, its address is used.

Definition 3.48 (thread address)**.** A *thread address* is given by a term of the grammar

$$a ::= \epsilon \mid a.g(i/s)$$

where ϵ is the empty address and $g, i, s \in \mathbb{N}_0$ are natural numbers. The set of all *thread addresses* is denoted by $\mathcal{T}$. Further, let $a \in \mathcal{T}$ be an address and $x \in \mathbb{N}_0$ be a natural number. The address $a[x]$ of the x-th parent is defined by

$$a[x] = \begin{cases} a & \text{if } x = 0 \\ a_1[x-1] & \text{if } a = a_1.g(i/s) \text{ and } x > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Also the *id* of a thread within its group is defined by

$$a.id = \begin{cases} \text{undefined} & \text{if } a = \epsilon \\ i & \text{if } a = a_1.g(i/s) \end{cases}$$

the *group size* is given by

$$a.size = \begin{cases} \text{undefined} & \text{if } a = \epsilon \\ s & \text{if } a = a_1.g(i/s) \end{cases}$$

and the *group id* by

$$a.group = \begin{cases} \text{undefined} & \text{if } a = \epsilon \\ g & \text{if } a = a_1.g(i/s) \end{cases}$$

Example 3.10 (thread address)**.** The address of a thread may be $\epsilon.0(1/2)$ where the group ID is 0, the thread's id is 1 and there are a total of 2 threads in its associated thread group. If this thread spawns a thread group 1 consisting of two threads the addresses of the new threads are $\epsilon.0(1/2).1(0/2)$ and $\epsilon.0(1/2).1(1/2)$. If the original thread $\epsilon.0(1/2)$ spawns a second thread group 2 consisting of three threads their addresses are $\epsilon.0(1/2).2(0/3)$, $\epsilon.0(1/2).2(1/3)$, and $\epsilon.0(1/2).2(2/3)$.

Definition 3.49 (program state)**.** The state of a program is defined by a pair

$$(e, s) \in (\mathcal{E} \times 2^{\mathcal{T} \times \mathbb{IS}}) = \mathcal{S}$$

where e is the current environment state of the execution and s a set of threads represented by their address and their current evaluation state formalized by an intermediate statement.

Program State Transitions

The operational semantic of INSPIRE is defined by the binary relation

$$\rightarrow \subset \mathcal{S} \times \mathcal{S}$$

and its transitive closure $\rightarrow^*$ as it is defined within the following sections. However, most transitions occur within individual threads without effecting others. Therefore an auxiliary relation

$$\rightarrow_s \subset (\mathcal{E} \times \mathcal{T} \times \mathbb{IS}) \times (\mathbb{IS} \times \mathbb{IS}) \times (\mathcal{E} \times \mathbb{IS})$$

is defined for modeling sequential transitions. As usual an infix notation is utilized for both relations.

Example 3.11 (sequential transition)**.** The utility of the relation $\rightarrow_s$ is best described by an example. The statement

$$(e, a, s) \xrightarrow{x \rightarrow x'}_s (e', s')$$

states that in environment e a thread with address a can conduct an operation $x \rightarrow x'$ replacing $x \in \mathbb{IS}$ by $x' \in \mathbb{IS}$ to transition from state s into state s'. By doing so, the environment is updated to e'.

Global State Transitions On a global level any thread may progress independently with the exception of synchronization operations. This is covered by the inference rule

$$\frac{(e, a_k, s_k) \xrightarrow{x \rightarrow x'}_s (e', s'_k) \qquad x \notin \mathbb{IS}_{gsync}}{(e, \{t_1, \ldots, (a_k, s_k), \ldots, t_n\}) \rightarrow (e', \{t_1, \ldots, (a_k, s'_k), \ldots, t_n\})} \text{ (step)}$$

The effect of synchronizing operations are covered in the following subsections. By repeatedly following the binary relation defined by those inference rules the threads covered in the program state are gradually resolved (=processed) until they reach an irreducible state. At this point they terminate their execution and are removed from the program state. This step is covered by

$$\frac{}{(e, \{\ldots, t_{k-1}, (a_k, \nu), t_{k+1}, \ldots\}) \rightarrow (e, \{\ldots, t_{k-1}, t_{k+1}, \ldots\})} \text{ (drain)}$$

where $\nu \in \mathbb{E}_i$ is the value the completed thread a_k has been reduced to[12].

[12]For the semantic formalization everything is an expression and every expression is reduced to a value.

Note that the (step) rule ensures *sequential consistency*. Hence, any environment obtained by applying a parallel program on some initial state can also be obtained by sequentially applying a sequence of the individual operations constraint by their partial order imposed by the processing threads and involved synchronization events. Other consistency models may be introduced by allowing multiple threads to progress concurrently – which requires their effects on the environment to be aggregated accordingly. For instance, let $\perp \in \mathcal{V}$ represent an *undefined value* and $\cup : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ be defined by

$$\epsilon_1 \cup \epsilon_2 = \begin{cases} \epsilon_1 & \text{if } \epsilon_2 = \epsilon \\ (\epsilon_1 \cup \epsilon_2')[x \mapsto y] & \text{if } \epsilon_2 = \epsilon_2'[x \mapsto y] \wedge x \notin \operatorname{dom}(\epsilon_1) \\ (\epsilon_1 \cup \epsilon_2')[x \mapsto y] & \text{if } \epsilon_2 = \epsilon_2'[x \mapsto y] \wedge x \in \operatorname{dom}(\epsilon_1) \wedge \epsilon_1[x] = y \\ (\epsilon_1 \cup \epsilon_2')[x \mapsto \perp] & \text{if } \epsilon_2 = \epsilon_2'[x \mapsto y] \wedge x \in \operatorname{dom}(\epsilon_1) \wedge \epsilon_1[x] \neq y \end{cases}$$

such that it is merging two environments by extending one by the key bindings of the other except for cases in which identical keys are bound to differing values. In such cases the corresponding keys will be bound to an undefined value in the resulting environment. Based on those, an inference rule like

$$\frac{\emptyset \neq I \subseteq \{1, \dots, n\} \qquad \forall_{i \in I}.(e, a_i, s_i) \xrightarrow{x_i \to x_i'}_s (e_i', s_i') \qquad \forall_{i \in I}.x_i \notin \mathbb{IS}_{gsync}}{(e, \{(a_1, s_1), \dots, (a_n, s_n)\}) \to (\bigcup_{i \in I} e_i', \bigcup_{i \notin I}\{(a_i, s_i)\} \cup \bigcup_{i \in I}\{(a_i, s_i')\})}$$

where I determines an arbitrary subset of threads to progress concurrently, models a system where concurrent, inconsistent environment modifications are supported, yet lead to undefined behavior. In particular, in the presence of *race conditions* the results of related operations are always undefined.

However, since the techniques and solutions presented in this and the following chapters are hardly effected by the details of the utilized consistency model, we will base our specification on the *sequential consistency* model constituted by the (step) rule for simplicity.

Definition 3.50 (traces)**.** The semantic of a program is defined by the set of traces it may follow during its execution. A trace is a sequence $[s_1, \dots, s_n] \in \mathcal{S}^*$ such that $s_i \to s_{i+1}$ for all $1 \leq i < n$. The set of traces $T(p)$ of a valid INSPIRE program $p \in \mathbb{P}$ is given by

$$T(p) = \{[s_1, \dots, s_n] \in \mathcal{S}^* \mid s_1 = (\epsilon, \{(\epsilon.0(0/1), \operatorname{conv}(p))\}) \wedge \forall_{i=1}^{n-1} s_i \to s_{i+1}\}$$

where $(\epsilon, \{(\epsilon.0(0/1), \operatorname{conv}(p))\}) \in \mathcal{S}$ is the initial state of p.

A program execution terminates if a final state $(e, \emptyset) \in \mathcal{S}$ for some environment e is reached.

With those inference rules the formalization of the overall IR semantic has been mostly reduced to the definition of the $\to_s$ relation which is covered in the following by distinguishing the individual expressions and statements.

Auxiliary Relations In many of the following cases several of the components of the modeled relations are just passed through from one side to the other or derived from the other components. To avoid overly excessive inference rule descriptions we define some auxiliary relations to reduce the complexity of the following inference rule specifications.

For transitions neither effected by the environment nor the address of the thread conducting it we introduce the auxiliary relation

$$\rightarrow_r \subset \mathbb{IS} \times \mathbb{IS}$$

which represents an isolated single reduction step from one intermediate statement to another. The bridge between this relation and the $\rightarrow_s$ relation is established by

$$\frac{s \rightarrow_r s'}{(e, a, s) \xrightarrow{s \rightarrow s'}_s (e, s')} \text{(r)}$$

Other transitions may depend on the current thread state and the threads address. For those cases we define the slightly extended relation

$$\rightarrow_a \subset (\mathcal{T} \times \mathbb{IS}) \times (\mathbb{IS})$$

which is linked to the $\rightarrow_s$ relation by

$$\frac{(a, s) \rightarrow_a (s')}{(e, a, s) \xrightarrow{s \rightarrow s'}_s (e, s')} \text{(a)}$$

Finally, some transition may effect the environment without depending on the thread address. For those we define the relation

$$\rightarrow_e \subset (\mathcal{E} \times \mathbb{IS}) \times (\mathcal{E} \times \mathbb{IS})$$

being connected to the $\rightarrow_s$ relation by

$$\frac{(e, s) \rightarrow_e (e', s')}{(e, a, s) \xrightarrow{s \rightarrow s'}_s (e', s')} \text{(e)}$$

Based on those definitions the semantic of individual IR language constructs is specified in the following sections.

Example 3.12 (auxiliary relations)**.** Let $s_1, s_2 \in \mathbb{IS}$ be two intermediate statements such that

$$s_1 \rightarrow_r s_2$$

can be proven by one of the following inference rules. Based on this we can prove for a given environment e and thread address a the validity of the transition

$$(e, a, s_1) \xrightarrow{s_1 \rightarrow s_2}_s (e, s_2)$$

and consequently the validity of e.g. the global step

$$(e, \{(a, s_1)\}) \to (e, \{(a, s_2)\})$$

by

$$\frac{\dfrac{s_1 \to_r s_2}{(e, a, s_1) \xrightarrow{s_1 \to s_2}_s (e, s_2)}\,\text{(r)} \qquad s_1 \notin \mathbb{IS}_{gsync}}{(e, \{(a, s_1)\}) \to (e, \{(a, s_2)\})}\,\text{(step)}$$

Provable steps of the $\to_a$ and $\to_e$ relations lead to corresponding global state transitions accordingly.

3.7.2 The Core Language Constructs

Expressions

We start by defining the semantic of expressions as they occur within *intermediate expressions* utilized for modeling the state of threads participating in the processing of an IR code fragment. We do so by elaborating the individual expression constructors – one after another.

Variables The first kind of expression is already a special case. Variables will never be reached by the execution since they get substituted right after their definition by their corresponding value. Hence, when entering a function all parameters (=variables) within the body are replaced by the argument values and whenever a variable is declared using a *declaration statement* the uses of the variable are substituted by the value the variable is bound to. Both situations will be covered in the context of the corresponding language constructs.

Literals The second kind of expression is the construct representing abstract values to be defined by the *interpretation* $I \in \mathcal{I}$. The single associated inference rule is given by

$$\frac{lit(i : t) \in \text{dom}(I)}{lit(i : t) : t \to_r I[lit(i : t)] : t}\,\text{(lit)}$$

simply stating that any literal can be reduced to its interpretation iff available. The given step is defined based on the $\to_r$ relation since it only depends on the incoming intermediate statement. As described above, this definition is inducing corresponding steps in the $\to_s$ relation and the global $\to$ state transition relation.

Example 3.13 (literals)**.** Let '5' $= \mathit{lit}(\text{'5'} : \mathit{int}\,\langle 4\rangle)$ be a literal and $I \in \mathcal{I}$ be the active interpretation such that $I[\text{'5'}] = 5 \in \mathbb{N} \subset \mathcal{V}$. Then we can prove the transition

$$\mathit{lit}(\text{'5'} : \mathit{int}\,\langle 4\rangle) : \mathit{int}\,\langle 4\rangle \rightarrow_r 5 : \mathit{int}\,\langle 4\rangle$$

and for instance a corresponding global transition

$$(e, \{(a, \mathit{lit}(5 : \mathit{int}\,\langle 4\rangle) : \mathit{int}\,\langle 4\rangle)\}) \rightarrow (e, \{(a, 5 : \mathit{int}\,\langle 4\rangle)\})$$

utilizing the inference rules (lit), (r) and (step).

Calls Call expressions are the first more complex operations since they are composed of sub-expressions which have to be evaluated first. Also additional steps regarding the proper instantiation of generic type parameters have to be conducted. The general order of processing calls is therefore covering three steps:

1. compute targeted function and arguments (in fixed order)
2. instantiate function body (if target function is a callable)
3. evaluate the function body (if target function is a callable)

The first step is about the evaluation of the target function and argument values. Those steps are covered by the inference rules

$$\frac{(e, a, e_0) \xrightarrow{op}_s (e', e_0')}{(e, a, e_0(e_1, \ldots, e_n) : t) \xrightarrow{op}_s (e', e_0'(e_1, \ldots, e_n) : t)} \text{(call trg)}$$

$$\frac{(e, a, e_i) \xrightarrow{op}_s (e', e_i')}{(e, a, \nu(\nu_1, \ldots, \nu_{i-1}, e_i, \ldots) : t) \xrightarrow{op}_s (e', \nu(\nu_1, \ldots, \nu_{i-1}, e_i', \ldots) : t)} \text{(call arg)}$$

where the first allows the target function to be processed while the second does the same for any of the involved arguments. As within all rules the operation *op* describes the replacement that has to be conducted somewhere within e.g. e_x to obtain e_x'. Note that the evaluation order of the targeted function and the involved arguments is fixed from left to right.

Finally, at some point the target function and its arguments will be completely evaluated – hence they have reached an irreducible state. A this point we have to conduct the actual function call which is covered by the two inference rules

$$\frac{\begin{array}{c}\sigma = \mathrm{MGS}([t_1, \ldots, t_n], [\mathrm{type}(\nu_1), \ldots, \mathrm{type}(\nu_n)]) \\ s' = \epsilon[v_1 : t_1 \mapsto \nu_1] \ldots [v_n : t_n \mapsto \nu_n](\sigma(s))\end{array}}{(((v_1 : t_1, \ldots, v_n : t_n) \rightarrow s)(\nu_1, \ldots, \nu_n) : t) \rightarrow_r \mathit{eval}(s') : t} \text{(call callable)}$$

and

$$\frac{\nu(e, \nu_1, \ldots, \nu_n) = (e', \nu') \qquad \nu : t_0 \text{ is not a callable}}{(e, \nu : t_0(\nu_1 : t_1, \ldots, \nu_n : t_n) : t) \rightarrow_e (e', \nu' : t)} \text{ (call value)}$$

where in the second rule $\nu_*, \nu' \in \mathcal{V}$ are values instead of the conventional irreducible expressions $\mathbb{IE}_i$.

The first covers the situation in which the targeted function has been reduced to a *callable* which has to be evaluated by instantiating the generic types within the body s followed by binding the callable's parameters to the values of the handed in arguments. The resulting *eval* construct is forming the scope for any potential return-value propagation as it is covered within the section regarding the *eval* construct.

The second rule, on the other hand, is applicable if the targeted function has evaluated into an actual function $\nu \in \mathcal{V}$, e.g. by interpreting a abstract symbol. In this case the function is evaluated based on the current environment and the list of arguments. The resulting modified environment and value $\nu' \in \mathcal{V}$ is used for restricting the allowed successor states within the $\rightarrow_s$ relation. This rule is the interface for the integration of any language extension to interact with the environment and the state of the processing thread.

Example 3.14 (call expression – callable object)**.** Let $f \in \mathbb{IE}$ be an intermediate expression reducible to the callable expression $(p, q) \rightarrow s : t$ where $p, q \in \mathbb{IV}$ are variables, $s \in \mathbb{IS}$ a statement forming a function body and $t \in \mathbb{T}$ the return type of the function f. Further, let $x, y \in \mathbb{IE}$ be two expressions reducible to x' and y'. Then for any environment $e \in \mathcal{E}$ and thread address $a \in \mathcal{T}$ we can prove the global steps

$$\begin{aligned}
(e, \{(a, f(x, y))\}) &\rightarrow^* (e, \{(a, ((p, q) \rightarrow s : t)(x, y))\}) && \text{(using (call trg))} \\
&\rightarrow^* (e, \{(a, ((p, q) \rightarrow s : t)(x', y))\}) && \text{(using (call arg))} \\
&\rightarrow^* (e, \{(a, ((p, q) \rightarrow s : t)(x', y'))\}) && \text{(using (call arg))} \\
&\rightarrow (e, \{(a, eval(s') : t)\}) && \text{(using (call callable))}
\end{aligned}$$

where s' is the function body s instantiated by the generic variable instantiation σ and having its formal parameters p and q replaced by its actual parameters x' and y' according to the inference rule (call callable).

Example 3.15 (call expression – abstract operator)**.** For comparison, let $f \in \mathbb{IE}$ be an abstract operator, hence a literal, representing the $+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ operator for natural numbers and '2', '3' $\in \mathbb{IS}$ constants representing corresponding integers. Further, let the interpretation $I \in \mathcal{I}$ include corresponding interpretations. Then for any environment $e \in \mathcal{E}$ and thread address

$a \in \mathcal{T}$ we can prove the global steps

$$\begin{aligned}(e, \{(a, f(\text{'2'}, \text{'3'}))\}) &\to (e, \{(a, +(\text{'2'}, \text{'3'}))\}) && \text{(using (call trg) and (lit))}\\ &\to (e, \{(a, +(2, \text{'3'}))\}) && \text{(using (call arg) and (lit))}\\ &\to (e, \{(a, +(2, 3))\}) && \text{(using (call arg) and (lit))}\\ &\to (e, \{(a, 5)\}) && \text{(using (call value))}\end{aligned}$$

where we omit the type annotations of intermediate expressions for clarity. The first step, for instance, is based on the inference tree

$$\dfrac{\dfrac{\dfrac{\dfrac{f \in \mathrm{dom}(I)}{f \to_r +}\,\text{(lit)}}{(e, a, f) \xrightarrow{f \to +}_s (e, +)}\,\text{(r)}}{(e, a, f(\text{'2'}, \text{'3'})) \xrightarrow{f \to +}_s (e, +(\text{'2'}, \text{'3'}))}\,\text{(call trg)} \qquad f \notin \mathbb{S}_{gsync}}{(e, \{(a, f(\text{'2'}, \text{'3'}))\}) \to (e, \{(a, +(\text{'2'}, \text{'3'}))\})}\,\text{(step)}$$

The remaining steps can be verified accordingly.

The given inference tree demonstrates the mechanics of the established relation infrastructure. The relation $\to_s$ is utilized for thread-local transitions. However, the actual reduction may be applied on some nested substructure, yet the global step needs to know whether the conducted operation is a synchronizing operation or not. Therefore the $\to_s$ relation is annotated by the actual modification $f \to +$ which is carried out to the global (step) rule where the corresponding property can be checked.

Structs and Unions *Struct* and *union* expressions evaluate their sub-expressions before packing the resulting values into a corresponding data structure. The evaluation of the sub-expressions is covered by the inference rules

$$\dfrac{(e, a, e_i) \xrightarrow{op}_s (e', e'_i) \qquad \forall_{1 \le x < i} \,.\, e_x \in \mathbb{E}_i}{(e, a, \mathit{struct}\,\{\ldots, n_i = e_i, \ldots\}) \xrightarrow{op}_s (e', \mathit{struct}\,\{\ldots, n_i = e'_i, \ldots\})}\,\text{(s\_arg)}$$

and

$$\dfrac{(e, a, e_i) \xrightarrow{op}_s (e', e'_i)}{(e, a, \mathit{union}\,\{n = e\}) \xrightarrow{op}_s (e', \mathit{union}\,\{n = e'\})}\,\text{(u\_arg)}$$

and the packing by

$$\dfrac{}{\mathit{struct}\,\{n_1 = \nu_1, \ldots, n_k = \nu_k\} \to_r s[n_1 : \nu_1, \ldots, n_k : \nu_k]}\,\text{(s\_value)}$$

and

$$\frac{}{\mathit{union}\,\{n = \nu : t\} \rightarrow_r u[\text{pack}_t(\nu)]}\ \text{(u\_value)}$$

where the terms of the shape $s[n_1 : \nu_1, \ldots, n_k : \nu_k] \in \mathcal{V}$ and $u[\nu] \in \mathcal{V}$ are the structures utilized for encoding struct and union values and the function $\text{pack}_t : \mathcal{V} \rightarrow \mathcal{V}$ packs a value of type t into a common data format for unions (e.g. a binary format). The reverse operation $\text{unpack}_t : \mathcal{V} \rightarrow \mathcal{V}$ is utilized to extract data from the resulting union. The details of this encoding are beyond the scope of this section. The only essential constraint at this point is that for all types $t \in \mathbb{T}_v$ and values $v \in D(t)$ in the domain of t the constraint

$$\text{unpack}_t(\text{pack}_t(v)) = v$$

is satisfied. Hence, a value stored within a union utilizing a specific type can be properly restored using the same type.

Member Access *Access expressions* are unpacking the values packed by struct and union expressions. The corresponding operation is covered by the inference rules

$$\frac{}{(s[\ldots, n_i : \nu_i, \ldots] : \mathit{struct}\ \{\ldots, n_i : t_i, \ldots\}).n_i : t_i \rightarrow_r \nu_i : t_i}\ \text{(access struct)}$$

and

$$\frac{\text{type}(\nu) = t_i}{(u[\nu] : \mathit{union}\ \{\ldots, n_i : t_i, \ldots\}).n_i : t_i \rightarrow_r \text{unpack}_{t_i}(\nu) : t_i}\ \text{(access union)}$$

where the first is extracting a value contained within a structure created by a struct expression and the second the value packed by a union expression.

Example 3.16 (accessing structs)**.** To demonstrate the interaction of the struct packing and the extraction of values we demonstrate the evaluation of the expression

$$(\mathit{struct}\ \{a = \text{'12'}; b = \text{'14'};\}).a$$

creating a struct and extracting the value of a member field. For any environment $e \in \mathcal{E}$ and thread address $a \in \mathcal{T}$ we can prove the global steps

$$\begin{aligned}
&(e, \{(a, (\mathit{struct}\ \{a = \text{'12'}; b = \text{'14'};\}).a)\}) \rightarrow \\
&\quad \rightarrow (e, \{(a, (\mathit{struct}\ \{a = 12; b = \text{'14'};\}).a)\}) && (\text{(s\_arg) and (lit)}) \\
&\quad \rightarrow (e, \{(a, (\mathit{struct}\ \{a = 12; b = 14;\}).a)\}) && (\text{(s\_arg) and (lit)}) \\
&\quad \rightarrow (e, \{(a, s[a : 12, b : 14].a)\}) && (\text{using (s\_value)}) \\
&\quad \rightarrow (e, \{(a, 12)\}) && (\text{using (access struct)})
\end{aligned}$$

where we omit the type annotations of intermediate expressions for clarity.

Example 3.17 (accessing unions)**.** For comparison we also demonstrate a similar case for a union expression given by the code fragment

$$(union\ \{a = \text{'12'}\}).a$$

creating a union value and extracting the contained field. Let *int* be the type of integer values within this example. For any environment $e \in \mathcal{E}$ and thread address $a \in \mathcal{T}$ we can prove the global steps

$$\begin{aligned}
&(e, \{(a, (union\ \{a = \text{'12'} : int\}).a : int)\}) \rightarrow \\
&\quad \rightarrow (e, \{(a, (union\ \{a = 12 : int\}).a : int)\}) && \text{((u\_arg) and (lit))} \\
&\quad \rightarrow (e, \{(a, u[\text{pack}_{int}(12)].a : int)\}) && \text{(using (u\_value))} \\
&\quad \rightarrow (e, \{(a, \text{unpack}_{int}(\text{pack}_{int}(12)))\}) && \text{(using (access union))} \\
&\quad = (e, \{(a, 12)\})
\end{aligned}$$

where we omit the type annotations of intermediate expressions for clarity except for the two essential annotations – the type of the value packed and the type expected as the result of the unpack operation. Those two operations are also the reason why type annotations had to be added to intermediate expression. Only the presence of those types enable the proper application of the corresponding pack and unpack conversions.

Recursive Function and Recursive Function Variables Function expressions are reduced to callable expressions by resolving recursive relations. Therefore a given (recursive) function is first unfolded to obtain the body to be processed when invoking the represented function. This body statement is then packed into a *callable* expression, together with the list of exposed variables.

Definition 3.51 (recursive function unfolding)**.** Let $f \in \mathbb{E}_v$ be the valid (recursive) function

$$rec\ f_x.\{f_1 = (v_{11}, \ldots, v_{1n_1}) \rightarrow t_1\ s_1, \ldots, f_k = (v_{k1}, \ldots, v_{kn_k}) \rightarrow t_k\ s_k\}$$

The unfolded statement unfold(f) $\in \mathbb{S}$ is obtained by replacing every unbound *recursive variable* f_j in s_x by

$$rec\ f_j.\{f_1 = (v_{11}, \ldots, v_{1n_1}) \rightarrow t_1\ s_1, \ldots, f_k = (v_{k1}, \ldots, v_{kn_k}) \rightarrow t_k\ s_k\}]$$

for all $1 \leq j \leq k$. Further, let $f' = \text{conv}(f) \in \mathbb{IE}$ be the annotated equivalent of f within the set of intermediate expressions. Then unfold(f') $\in \mathbb{IS}$ is defined by conv(unfold(f)).

Based on the unfolded function body the reduction of a *function expression* into a *callable intermediate expression* is incorporated by

$$\frac{s' = \text{unfold}(rec\ f_x.\{\ldots, f_x = (v_1,\ldots,v_n) \to t\ s, \ldots\} : t)}{rec\ f_x.\{\ldots, f_x = (v_1,\ldots,v_n) \to t\ s, \ldots\} : t \to_r (v_1,\ldots,v_n) \to s' : t}\ \text{(fun)}$$

Since the unfolding process is not exposing unbound recursive function variables and due to the fact that no valid IR fragment is exhibiting any free recursive function variables, recursive variables will never by reached during the resolution process. Hence no rules have to be specified for those.

The reason for converting functions into *callables* is to provide a uniform construct for any callable object in the IR. Since the value of *bind* expressions is also a potential target for calls a similar reduction has to be offered for those too.

Bind Before turning a bind into a *callable* expression its captured values have to be evaluated. This is driven by the two inference rules

$$\frac{(e, a, e_i) \xrightarrow{op}_s (e', e_i')}{(e, a, (\ldots) \Rightarrow e_0(e_1,\ldots,e_n) : t) \xrightarrow{op}_s (e', (\ldots) \Rightarrow e_0'(e_1,\ldots,e_n) : t)}\ \text{(bind trg)}$$

and

$$\frac{(e, a, e_i) \xrightarrow{op}_s (e', e_i') \qquad \forall 0 \le j < i\ .\ e_j \in (\mathbb{IV} \cup \mathbb{IE}_i)}{(e, a, (\ldots) \Rightarrow e_0(\ldots, e_i, \ldots) : t) \xrightarrow{op}_s (e', (\ldots) \Rightarrow e_0(\ldots, e_i', \ldots) : t)}\ \text{(bind arg)}$$

where the first is focusing on the target function of the nested call expression while the latter is handling its arguments. However, since we have no transition rule for variables we will eventually end up with a nested call expression consisting of irreducible expressions and free variables – those to be left unbound within the resulting *callable* expression. At this state we can close the *bind* expression by reducing it into a *callable* expression. This step is covered by

$$\frac{\forall_{j=0}^{n}\ .\ e_j \in (\mathbb{IE}_i \cup \{v_1,\ldots,v_k\})}{(v_1,\ldots,v_k) \Rightarrow e_0(e_1,\ldots,e_n) : t \to_r (v_1,\ldots,v_k) \to e_0(e_1,\ldots,e_n) : t}\ \text{(bind)}$$

which ensured that it can only be applied once all captured values have been fully evaluated. In the end both, *function* and *bind* expressions are reduced to *callable* expressions which can be utilized as the target of a call expression.

Job This kind of expression has several sub-expressions whose evaluation is covered by the three rules

$$\frac{(e, a, e_l) \xrightarrow{op}_s (e', e_l')}{(e, a, job\ [e_l, e_u]\ b : t) \xrightarrow{op}_s (e', job\ [e_l', e_u]\ b : t)}\ \text{(job low)}$$

$$\frac{(e, a, e_u) \xrightarrow{op}_s (e', e'_u)}{(e, a, job\ [\nu_l, e_u]\ b : t) \xrightarrow{op}_s (e', job\ [\nu_l, e'_u]\ b : t)}\ \text{(job up)}$$

$$\frac{(e, a, b) \xrightarrow{op}_s (e', b')}{(e, a, job\ [\nu_l, \nu_u]\ b : t) \xrightarrow{op}_s (e', job\ [\nu_l, \nu_u]\ b' : t)}\ \text{(job body)}$$

When the lower and upper boundary as well as the job body have reached an irreducible state a job can be closed to form a spawnable object by

$$\frac{}{(job\ [\nu_l, \nu_u]\ \nu_b : t) \rightarrow_r ([\nu_l, \nu_u] \rightarrow \nu_b : t)}\ \text{(job close)}$$

The resulting spawnable object combines constraints on the size of the thread group required to process the represented job and the job-body ν_b to be evaluated by processing threads.

Values, Callables and Spawnables This kind of intermediate expressions are irreducible expressions. Consequently there are no inference rules covering those. They serve as structures for case distinctions within other rules, in particular the rules covering *call expressions*.

Evaluate The *evaluation* construct is limiting the scope of return statements. An evaluation expression $eval(s) : t \in \mathbb{E}$ is evaluating the contained statement $s \in \mathbb{S}$ until it either evaluates to an irreducible expression (=a value) or to a *return statement* carrying an irreducible expression. In both cases the obtained value will be the value the *evaluation* expression is reduced to.

The gradual evaluation of the contained statement is covered by the rule

$$\frac{(e, a, s) \xrightarrow{op}_s (e', s')}{(e, a, eval(s) : t) \xrightarrow{op}_s (e', eval(s') : t)}\ \text{(eval step)}$$

the forwarding of an irreducible value by

$$\frac{\text{type}(\nu) = t}{eval(\nu) : t \rightarrow_r \nu}\ \text{(eval value)}$$

and the termination of a return stmt by

$$\frac{\text{type}(\nu) = t}{eval(return\ \nu) : t \rightarrow_r \nu}\ \text{(eval return)}$$

Note that the type safety constraints regarding the result type of the evaluation could be omitted since those are already enforced by the restriction to valid program fragments.

The *evaluation* expression is the cross-over point between the evaluation of expressions and statements. The semantic of those is covered next.

Statements

The processing of statements is driven by the stepwise execution of sequences of statements. The central construct for sequencing statements is the compound statement.

Compound When processing a *compound statement* consisting of several statements only the first statement is allowed to progress. This is covered by the rule

$$\frac{(e, a, s_0) \xrightarrow{op}_s (e', s'_0)}{(e, a, \{s_0; s_1; \ldots; s_n\}) \xrightarrow{op}_s (e', \{s'_0; s_1; \ldots; s_n\})} \text{(comp expr)}$$

Only if the first statement has reached an irreducible state it is dropped to enable the processing of the subsequent statement. The corresponding transition is incoorperated by

$$\frac{}{\{\nu; s_1; \ldots; s_n\} \rightarrow_r \{s_1; \ldots; s_n\}} \text{(comp step)}$$

Finally, if no more statements are left the resulting empty compound statement is reduced to the unit value $unit \in \mathcal{V}$ by

$$\frac{}{\{\} \rightarrow_r unit} \text{(comp unit)}$$

The *unit* value is the value all statements (not being expressions) are reduced to. It indicates that the conducted computation was not obtaining any value but may have been causing side effects by manipulating the environment.

Declaration A variable declaration is processed by evaluating the value the declared variable should be bound to. This is covered by

$$\frac{(e, a, e_0) \xrightarrow{op}_s (e', e'_0)}{(e, a, decl\ v = e_0) \xrightarrow{op}_s (e', decl\ v = e'_0)} \text{(decl expr)}$$

Once the evaluation has reached an irreducible state the declared variable is substituted within its scope by the obtained value. This step is covered by

$$\frac{\sigma = [v \mapsto \nu]}{\{decl\ v = \nu; s_1; \ldots; s_n\} \rightarrow_r \{\sigma(s_1); \ldots; \sigma(s_n)\}} \text{(decl value)}$$

where $\sigma(s_i) = [v \mapsto \nu](s_i)$ is replacing every free occurrence of the variable $v \in \mathbb{V}$ in $s_i \in \mathbb{S}$ by the irreducible expression $\nu \in \mathbb{E}_i$.

Condition A conditional statement starts by processing the condition expression and, based on the obtained result, one of the two branches follows. The first step is covered by

$$\frac{(e, a, e_0) \xrightarrow{op}_s (e', e_0')}{(e, a, if\ (e_0)\ then\ s_1\ else\ s_2) \xrightarrow{op}_s (e', if\ (e_0')\ then\ s_1\ else\ s_2)}\ \text{(if expr)}$$

and the branch selection by

$$\frac{}{if\ (true : bool)\ then\ s_1\ else\ s_2 \rightarrow_r s_1}\ \text{(if true)}$$

and

$$\frac{}{if\ (false : bool)\ then\ s_1\ else\ s_2 \rightarrow_r s_2}\ \text{(if false)}$$

where $true$ and $false$ are elements of the value set $\mathcal{V}$.

While The behavior of the while loop is based on the definition of the conditional statement above. By introducing

$$\frac{}{while\ (e_0)\ do\ s \rightarrow_r if\ (e_0)\ then\ \{loop(s); while\ (e_0)\ do\ s\}\ else\ \{\}}\ \text{(while)}$$

the semantic of a while loop is covered by relying on the rules established for conditional expressions. The $loop(s)$ construct enclosing the processed loop body is utilized to limit the scope of *break* and *continue* expressions analogous to the way *eval* limits the scope of *return* statements.

Note that for the given definition the condition expression e_0 is repeatedly evaluated before every single iteration of the while loop as well as once before the while loop is terminated – as desired.

For Unlike for while loops the range-boundaries of for loops are evaluated only once before entering the first iteration of the loop. The corresponding evaluation steps are covered by the three inference rules

$$\frac{(e, a, e_s) \xrightarrow{op}_s (e', e_s')}{\begin{array}{c}(e, a, for\ \ (v = e_s \ldots e_e : e_i)\ \ do\ s) \\ \xrightarrow{op}_s \\ (e', for\ \ (v = e_s' \ldots e_e : e_i)\ \ do\ s))\end{array}}\ \text{(for start)}$$

$$\frac{(e, a, e_e) \xrightarrow{op}_s (e', e_e')}{\begin{array}{c}(e, a, for\ \ (v = \nu_s \ldots e_e : e_i)\ \ do\ s) \\ \xrightarrow{op}_s \\ (e', for\ \ (v = \nu_s \ldots e_e' : e_i)\ \ do\ s))\end{array}}\ \text{(for end)}$$

$$\frac{(e,a,e_i) \xrightarrow{op}_s (e',e_i')}{\begin{array}{c}(e,a,for\ \ (v=\nu_s\ldots\nu_e : e_i)\ \ do\ s)\\ \xrightarrow{op}_s \\ (e',for\ \ (v=\nu_s\ldots\nu_e : e_i')\ \ do\ s))\end{array}} \text{(for inc)}$$

The actual loop iterations are then covered by

$$\frac{\nu_s,\nu_e,\nu_i \in \mathbb{Z} \qquad (\nu_i > 0 \wedge \nu_s < \nu_e) \vee (\nu_i < 0 \wedge \nu_s > \nu_e)}{\begin{array}{c}for\ \ (v=\nu_s\ldots\nu_e : \nu_i)\ \ do\ s\\ \rightarrow_r \\ \{loop(\{decl\ v=\nu_s; s\}); for\ \ (v=\nu_s+\nu_i\ldots\nu_e : \nu_i)\ \ do\ s\}\end{array}} \text{(for step)}$$

after the evaluation of the range-boundaries has reached an irreducible state. The iterator variable v is incorporated by utilizing the inference rules covering declaration statements and as for the while loop a *loop* construct is introduced to limit the scope of *continue* statements[13]. Finally the termination of the loop is covered by

$$\frac{\nu_s,\nu_e,\nu_i \in \mathbb{Z} \qquad \neg((\nu_i > 0 \wedge \nu_s < \nu_e) \vee (\nu_i < 0 \wedge \nu_s > \nu_e))}{for\ \ (v=\nu_s\ldots\nu_e : \nu_i)\ \ do\ s \rightarrow_r \{\}} \text{(for done)}$$

Note that in any case the range-boundaries are evaluated only once before entering the first loop iteration and can therefore not be influenced by side-effects caused by processing the loop body.

Return The processing of *return* statements is separated into two phases. In the first the value to be returned has to be evaluated using

$$\frac{(e,a,e_0) \xrightarrow{op}_s (e',e_0')}{(e,a,return\ e_0) \xrightarrow{op}_s (e',return\ e_0')} \text{(return expr)}$$

Once this evaluation is completed the return statement starts escaping enclosing compound statements by following the transitions validated by

$$\frac{}{\{return\ \nu; s_1; \ldots; s_n\} \rightarrow_r return\ \nu} \text{(comp return)}$$

until an $eval(\ldots)$ expression is reached and the computed return value is delivered to the call site of the corresponding callable.

[13]Break statements are prohibited by the definition of valid IR fragments.

Break, Continue and Loop Break and continue statements are consuming subsequent statements within the compound statement due to the rules

$$\frac{}{\{break; s_1; \ldots; s_n\} \rightarrow_r break} \text{(break prop)}$$

and

$$\frac{}{\{continue; s_1; \ldots; s_n\} \rightarrow_r continue} \text{(continue prop)}$$

until they reach a loop statement. At the loop statement the rules

$$\frac{}{\{loop(continue); s\} \rightarrow_r s} \text{(loop continue)}$$

and

$$\frac{}{\{loop(break); s\} \rightarrow_r \{\}} \text{(loop break)}$$

determine whether the next iteration of a loop (represented by s – which is either a while or for loop statement) is processed (continue) or the loop is terminated (break). Further, in case the sub-statement of a loop statement is evaluation to a *return* statement the return is passed through by

$$\frac{}{loop(return\ \nu) \rightarrow_r return\ \nu} \text{(loop ret)}$$

However, in case no *break*, *continue*, or *return* has yet reached the *loop* statement, the inner body is allowed to progress as defined in

$$\frac{(e, a, s) \xrightarrow{op}_s (e', s')}{(e, a, loop(s)) \xrightarrow{op}_s (e', loop(s'))} \text{(loop step)}$$

and if this evaluation reaches a irreducible state the full loop body evaluates to this value due to

$$\frac{}{loop(\nu) \rightarrow_r \nu} \text{(loop iter)}$$

Parallel Constructs

In a last step the semantic of the parallel constructs offered by INSPIRE is covered by the following set of inference rules.

Thread Identification To access information on a processing thread, in particular its id and the size of its associated thread group, the functions *getThreadID* and *getNumThreads* have been specified within the syntax section. Their semantic is defined by the two inference rules[14]

$$\frac{\nu \in \mathbb{N}_0}{(a, getThreadID(\nu)) \rightarrow_a (a[\nu].id)} \text{ (thread id)}$$

and

$$\frac{\nu \in \mathbb{N}_0}{(a, getNumThreads(\nu)) \rightarrow_a (a[\nu].size)} \text{ (group size)}$$

Both rules utilize the locally available thread address to obtain the required information.

Spawn The creation of a thread is a local operation to be conducted by individual threads causing global effects. For the local step we introduce the inference rule

$$\frac{g \in \mathbb{N}_0 \text{ is a fresh group ID}}{spawn([\nu_l, \nu_u] \rightarrow b) \rightarrow_r g} \text{ (local spawn)}$$

enabling arbitrary threads to process spawn operations. However, since a call $spawn(..) \in \mathbb{S}_{gsync}$ is a global synchronizing expression the general rule (step) introduced on page 95 is not applicable to close the gap between the $\rightarrow_s$ relation and the global system state transition relation $\rightarrow$. Instead this gap is closed by

$$\frac{(e, t, s) \xrightarrow{spawn([\nu_l, \nu_u] \rightarrow b) \rightarrow g}_s (e, s') \qquad \nu_l \leq n \leq \nu_u}{(e, \{\ldots, (t, s), \ldots\}) \rightarrow (e, \{\ldots, (t, s'), \ldots\} \cup \bigcup_{i=0}^{n-1} \{(t.g(i/n), b())\})} \text{ (spawn)}$$

which is considering the global side-effects of the local spawn operation processed within a single thread. The rule states that whenever there is a thread t capable of evolving from a state s to a state s' by processing a *spawn* operation it is allowed to do so. As a side effect n threads, where n is between the lower and upper group-size boundaries of the *spawnable* $[\nu_l, \nu_u] \rightarrow b$, are created. The addresses of the new threads are extensions of the spawning thread and each of the new threads is processing the job-body b.

[14]This works under the assumption that $getThreadID, getNumThreads \notin \text{dom}(I)$.

Merge To specify the semantic of the *merge* operations we utilize the same pattern as for the *spawn* operation. The reduction step

$$\frac{}{merge(\nu) \rightarrow_r unit} \text{(local merge)}$$

enables local threads to progress. However, since as for the spawn operation $merge(\ldots)$ is a global synchronizing expression the general (step) rule is not applicable. Instead the rule

$$\frac{(e, a_i, s_i) \xrightarrow{merge(\nu) \rightarrow unit}_s (e, s_i') \qquad \nexists j \;.\; a_j.group = \nu}{(e, \{\ldots, (a_i, s_i), \ldots\}) \rightarrow (e, \{\ldots, (a_i, s_i'), \ldots\})} \text{(merge)}$$

is introduced ensuring that the state transitions induced by the local merge operations are only enabled if all threads of the corresponding group have terminated their execution and have been evicted from the thread pool.

A variation of this schema is provided for the merge-all operation whose semantic is fixed by the local transition

$$\frac{}{merge() \rightarrow_r unit} \text{(local merge all)}$$

and the global step

$$\frac{(e, a_i, s_i) \xrightarrow{merge() \rightarrow unit}_s (e, s_i') \qquad \nexists j \;.\; \exists l \geq 1 \;.\; a_j[l] = a_i}{(e, \{\ldots, (a_i, s_i), \ldots\}) \rightarrow (e, \{\ldots, (a_i, s_i'), \ldots\})} \text{(merge all)}$$

While the former merge operator allows every thread to block until an arbitrary groups identified by a group id g has terminated its execution, the latter is blocking until all directly or indirectly spawned sub-threads have completed their processing.

Work Distribution Within the syntax section the operator

$$pfor : (int\langle 4\rangle, int\langle 4\rangle, int\langle 4\rangle, (int\langle 4\rangle, int\langle 4\rangle, int\langle 4\rangle) \Rightarrow unit) \rightarrow unit$$

has been introduced to distribute workload within a thread group. Threads within a thread group utilize this construct by passing (matching) ranges of processing steps (first three arguments) and a function capable of processing sub-ranges of the full range. The $pfor$ primitive will then coordinate the partitioning of the given range among the participating threads and cause them to process their assigned sub-ranges.

To formally clarify the operator's semantic *ranges* have to be defined.

Definition 3.52 (ranges)**.** Let $a, b, c \in \mathbb{Z}$ be three integers and $c \neq 0$. A *range* denoted by $a \ldots b : c$ is an abbreviation of the set

$$\{x \in \mathbb{Z} \mid a \leq x \wedge x < b \wedge \exists i \in \mathbb{N}_0 \;.\; x = a + ic\}$$

if $c > 0$ and

$$\{x \in \mathbb{Z} \mid a \geq x \wedge x > b \wedge \exists i \in \mathbb{N}_0 \,.\, x = a + ic\}$$

if $c < 0$ where a and b are the boundaries and c is the step size. Let the set of all ranges be denoted by $\mathcal{R}$. A *range partition* of a range $(a \ldots b : c) \in \mathcal{R}$ is a set of non-empty ranges $\{r_1, \ldots, r_n\} \in 2^{\mathcal{R}}$ such that $\bigcup_{i=1}^{n} r_i = (a \ldots b : c)$ and for each element x in $a \ldots b : c$ there is exactly one r_i such that $x \in r_i$.

To stipulate the partitioning of ranges we introduce a function

$$\text{shares} : (\mathcal{T} \times \mathbb{N} \times \mathcal{R}) \to 2^{\mathcal{R} \setminus \emptyset}$$

such that shares(a, p, r) assigns each thread a a set of sub-ranges of r for its p-th execution of a $pfor$ operation. For this function we have to ensure that the full range r is distributed among the members of a's thread group during their p-th $pfor$ evaluation.

Let $\text{shares}' : (\mathbb{N}^4 \times \mathcal{R}) \to 2^{\mathcal{R} \setminus \emptyset}$ be a function such that for all ranges $r \in \mathcal{R}$ and for all $g, s, p \in \mathbb{N}_0$ where $s > 0$ the set

$$\bigcup_{i=0}^{s-1} \text{shares}'(g, i, s, p, r)$$

is a range partition of r. Then we define $\text{shares} : (\mathcal{T} \times \mathbb{N} \times \mathcal{R}) \to 2^{\mathcal{R} \setminus \emptyset}$ by

$$\text{shares}(a, p, r) = \text{shares}'(a.group, a.id, a.size, p, r)$$

to obtain the desired properties.

To count the number of processed $pfor$ operations within a thread we introduce the counter $\text{pfc} : \mathcal{T} \to \mathbb{N}_0$ with the default value $\text{pfc}_\perp = 0$ to the environment state (see page 90).

The semantic of $pfor$ is then covered by

$$\frac{\begin{array}{c}\{(\nu'_{1s} \ldots \nu'_{1e} : \nu'_{1i}), \ldots, (\nu'_{ns} \ldots \nu'_{ne} : \nu'_{ni})\} = \text{shares}(a, \text{pfc}_e(a), \nu_s \ldots \nu_e : \nu_i) \\ op = pfor(\nu_s, \nu_e, \nu_i, \nu_b) \to \{\nu_b(\nu'_{1s}, \nu'_{1e}, \nu'_{1i}); \ldots ; \nu_b(\nu'_{ns}, \nu'_{ne}, \nu'_{ni})\} \\ e' = \text{pfc}_e(a){+}{+}\end{array}}{(e, a, pfor(\nu_s, \nu_e, \nu_i, \nu_b)) \xrightarrow{op}_s (e', \{\nu_b(\nu'_{1s}, \nu'_{1e}, \nu'_{1i}); \ldots ; \nu_b(\nu'_{ns}, \nu'_{ne}, \nu'_{ni})\})} \text{(pfor)}$$

reducing the local call $pfor(\nu_s, \nu_e, \nu_i, \nu_b)$ to the statement

$$\{\nu_b(\nu'_{1s}, \nu'_{1e}, \nu'_{1i}); \ldots ; \nu_b(\nu'_{ns}, \nu'_{ne}, \nu'_{ni})\}$$

processing the sub-ranges assigned to the local thread.

Note that the processing of $pfor$ is a thread-local operation and does not cause any synchronization. In particular some threads within a group may proceed processing subsequent $pfor$ calls before others may have finished earlier invocations. Also, $pfor$ operations are not *syntactically* linked – hence different $pfor$ instances within an IR code fragment being processed by threads within a common group may distribute workload among each other. The connection is established by the $pfor$ execution counter.

Finally it has to be pointed out that due to the specification of $pfor$ the full range of elements to be processed is only guaranteed to be covered if every thread within a group is passing identical ranges to connected $pfor$ operation invocations.

Example 3.18 (pfor application)**.** Let $a \in \mathcal{T}$ be a thread address and $a_1 = a.g(0/2) \in \mathcal{T}$ and $a_2 = a.g(1/2) \in \mathcal{T}$ the addresses of two threads constituting a thread group g spawned by a. Furthermore, let the function shares : $(\mathcal{T} \times \mathbb{N} \times \mathcal{R}) \to 2^{\mathcal{R} \setminus \emptyset}$ be defined such that

$$\text{shares}(a_1, 14, [0 \dots 10 : 1]) = \{[0 \dots 5 : 1], [9 \dots 10 : 1]\}$$

and

$$\text{shares}(a_2, 14, [0 \dots 10 : 1]) = \{[5 \dots 9 : 1]\}$$

Hence, the full range $[0 \dots 10 : 1]$ for the 14-th pfor invocation of thread group g is distributed among the two threads such that the first is responsible for the sub-ranges $[0 \dots 5 : 1]$ and $[9 \dots 10 : 1]$ while the second is responsible for the sub-range $[5 \dots 9 : 1]$. Other instantiations of the function shares lead to different workload distributions. However, by definition, there is no valid instantiation such that overlapping sub-ranges may be assigned to the involved threads nor may elements of the full range be skipped.

Let $e \in \mathcal{E}$ be an environment such that $\text{pfc}_e(a_1) = 14$, $s, s_2 \in \mathbb{IS}$ be arbitrary intermediate statements and $f \in \mathbb{IE}_i$ be a irreducible expression forming a valid body argument for a pfor invocation. Then we can prove the global transition

$$\begin{aligned}&(e, \{(a, s), (a_1, pfor(0, 10, 1, f)), (a_2, s_2)\}) \to \\ &\quad \to (e', \{(a, s), (a_1, \{f(0, 5, 1); f(9, 10, 1); \}), (a_2, s_2)\})\end{aligned}$$

utilizing the (pfor) inference rule where $e' = \text{pfc}_e(a_1){+}{+}$, hence a copy of the environment e where the counter pfc for thread a_1 has been increased by one. Equally we can prove the independent global transition

$$\begin{aligned}&(e_2, \{(a, s), (a_1, s_1), (a_2, pfor(0, 10, 1, g))\}) \to \\ &\quad \to (e_2', \{(a, s), (a_1, s_1), (a_2, \{g(5, 9, 1); \})\})\end{aligned}$$

for environments $e_2, e_2' \in \mathbb{E}$ such that $\text{pfc}_{e_2}(a_2) = 14$, $\text{pfc}_{e_2'}(a_2) = 15$ where $s, s_1 \in \mathbb{IS}$ are arbitrary intermediate statements and $g \in \mathbb{IE}_i$ is an irreducible expression representing a function capable of processing sub-ranges of the overall $pfor$ range in the context of a_2. Furthermore, due to the (step) inference rule regarding global transitions, non-synchronizing individual transitions within threads may be arbitrarily interleaved. Consequently, since $pfor$ is a non-synchronizing operation the processing of the sub-ranges may be conducted independently in any overlapping or even non-overlapping fashion.

Data Distribution The syntax section has also introduced the *data sharing* construct

$$redistribute : (\alpha, (array\,\langle\alpha\rangle) \Rightarrow \beta) \rightarrow \beta$$

The general idea is that every thread within a group is contributing a value of type α as the first argument and a projection function as a second argument selecting the data from the full list of contributed elements to be made locally available.

Since *redistribute* is a globally synchronizing operation we have to add support for the thread local step

$$\frac{\nu_1, \ldots, \nu_n \in \mathbb{E}_i}{redistribute(\nu_v, \nu_s) \rightarrow_r \nu_s([\nu_1, \ldots, \nu_n])}\ \text{(local redist)}$$

and the associated global transition

$$\frac{\forall_{i=0}^{n-1}(e, a.g(i/n), s_i) \xrightarrow{redistribute(\nu_{vi},\nu_{si}) \rightarrow \nu_{si}([\nu_{v1},\ldots,\nu_{vn-1}])}_s (e, s_i')}{\begin{array}{c}(e, \{\ldots, (a.g(0/n), s_0), \ldots, (a.g(n-1/n), s_{n-1}), \ldots\}) \\ \rightarrow \\ (e, \{\ldots, (a.g(0/n), s_0'), \ldots, (a.g(n-1/n), s_{n-1}'), \ldots\})\end{array}}\ \text{(redist)}$$

The local transition does not enforce any constraint on the resulting value. However, the global rule is restricting transitions to cases in which the exchanged data is consistent among the full list of involved threads. Those threads are connected via their group id.

From the provided definition it can be derived that the redistribution operation is a blocking operation causing every participating thread to wait until all threads within the group are ready to conduct the data exchange. Once this state is reached, all threads within the group are advanced simultaneously.

Example 3.19 (redistribute application)**.** Let $a \in \mathcal{T}$ be a thread address and $a_1 = a.g(0/2) \in \mathcal{T}$ and $a_2 = a.g(1/2) \in \mathcal{T}$ the addresses of two threads constituting a thread group g spawned by a. Further, let $f, g \in \mathbb{E}_i$ be irreducible expressions to be utilized as a second argument for the *redistribute* function. Then, for an arbitrary environment $e \in \mathcal{E}$ we can prove the transition

$$\begin{aligned}&(e, \{(a_1, redistribute(12, f)), (a_2, redistribute(14, g))\}) \rightarrow \\ &\quad \rightarrow (e, \{(a_1, f([12, 14])), (a_2, g([12, 14]))\})\end{aligned}$$

since $a_1 = a.g(0/2)$ and $a_2 = a.g(1/2)$ are all threads of group g and in the initial state they are both ready for the synchronized global transition and the included data exchange. The corresponding inference tree is given by

$$\frac{\text{Premise A} \qquad\qquad\qquad \text{Premise B}}{\begin{array}{c}(e, \{(a_1, r(12, f)), (a_2, r(14, g))\}) \\ \to \\ (e, \{(a_1, f([12, 14])), (a_2, g([12, 14]))\})\end{array}}\ \text{(redist)}$$

where $r = \mathit{redistribute}$ and Premise A is given by

$$\frac{\dfrac{12, 14 \in \mathcal{V}}{r(12, f) \to_r f([12, 14])}\ \text{(local redist)}}{(e, a_1, r(12, f)) \xrightarrow{r(12,f)\to f([12,14])}_s (e, f([12, 14]))}\ \text{(r)}$$

and Premise B is given by

$$\frac{\dfrac{12, 14 \in \mathcal{V}}{r(14, g) \to_r g([12, 14])}\ \text{(local redist)}}{(e, a_2, r(14, g)) \xrightarrow{r(14,g)\to g([12,14])}_s (e, g([12, 14]))}\ \text{(r)}$$

Note the interaction between the local transition rule (local redist) and the global transition rule (redist). The local transition would allows any thread at any time to progress by picking arbitrary values for the reduction. Hence, local transitions like

$$r(12, f) \to_r f([3, 12, 8, 9])$$

would also be supported by the $\to_r$ relation. However, those are not affecting the global state transition relation $\to$ since the conventional relation adapter, the (step) inference rule (see page 95), excludes reductions on global synchronizing operations including *redistribute* invocations. Hence, only local reductions validated by the global perspective of the (redist) inference rule are accepted and may hence cause a global state transition as demonstrated in this example.

Channels The last parallel element of INSPIRE is the channel infrastructure. A channel is a fixed length blocking queue which is utilized by threads to conduct synchronized point-to-point communication. To cover the behavior of channels, constructs to model their states are required.

Definition 3.53 (channel state)**.** The state of a channel is defined by a pair

$$(c, q) \in \mathbb{N}_+ \times \mathcal{V}^*$$

where $\mathbb{N}_+$ is the set of natural numbers not including 0, the element $c \in \mathbb{N}_+$ defines the capacity of the channel and the string q its current queue state. The length $|q|$ of a value string $q = [\nu_1, \ldots, \nu_n] \in \mathcal{V}^*$ is defined by

$$|[\nu_1, \ldots, \nu_n]| = n$$

and the set of all valid channel states $\mathcal{C}$ is defined by

$$\mathcal{C} = \{(c, q) \in \mathbb{N}_+ \times \mathcal{V}^* \mid |q| <= c\}$$

Let $s = (c, q) \in \mathcal{C}$ be a channel state. The functions empty : $\mathcal{C} \to \mathbb{B}$ defined by $(c, q) \mapsto (|q| = 0)$ and full : $\mathcal{C} \to \mathbb{B}$ defined by $(c, q) \mapsto (|q| = c)$ test whether a channel is *empty* or *full*. Further we define the partial function read : $\mathcal{C} \to \mathcal{V}$ by

$$\mathrm{read}((c, q)) := \begin{cases} \nu_n & \text{if } q = [\nu_1, \ldots, \nu_n] \text{ and } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

the partial function pop : $\mathcal{C} \to \mathcal{C}$ by

$$\mathrm{pop}((c, q)) := \begin{cases} (c, [\nu_1, \ldots, \nu_{n-1}]) & \text{if } q = [\nu_1, \ldots, \nu_n] \text{ and } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the partial function put : $(\mathcal{C} \times \mathcal{V}) \to \mathcal{C}$ by

$$\mathrm{put}((c, q), \nu) := \begin{cases} (c, [\nu, \nu_1, \ldots, \nu_n]) & \text{if } q = [\nu_1, \ldots, \nu_n] \text{ and } n < c \\ \text{undefined} & \text{otherwise} \end{cases}$$

Let $\mathcal{C}_i$ be an arbitrary set of *channel identifiers* and chl : $\mathcal{C}_i \to \mathcal{C}$ a function mapping identifiers to their state. Then the inference rule

$$\frac{c \in \mathcal{C}_i \text{ is a fresh channel id}}{(e, \mathit{channel.create}(\nu_t, \nu_c)) \to_e (\mathrm{chl}_e(c) \leftarrow (\nu_c, []), c)} \text{ (chl create)}$$

defines a thread-local operation creating a new empty channel to transfer values of type ν_t using a buffer with a capacity of $\nu_c \in \mathbb{N}_0$. The channel state itself is stored within the global environment. The release operation is covered by

$$\frac{}{(e, \mathit{channel.release}(\nu_c)) \to_e (e, \mathit{unit})} \text{ (chl release)}$$

Since for the specification of the semantic of IR constructs we are not concerned with resource consumption the fact that the channel state is not actually removed from the environment is ignored at this point. Nevertheless, the release operation is required for the resource management within actual implementations.

The operations sending/receiving messages to/from channels are covered by the inference rules

$$\frac{\neg\, \mathrm{full}(\mathrm{chl}_e(\nu_c))}{(e, \mathit{channel.send}(\nu_c, \nu_v)) \to_e (\mathrm{put}(\mathrm{chl}_e(\nu_c), \nu_v), \mathit{unit})} \text{ (chl send)}$$

and

$$\frac{\neg\,\text{empty}(\text{chl}_e(\nu_c))}{(e, \mathit{channel.recv}(\nu_c)) \to_e (\text{pop}(\text{chl}_e(\nu_c)), \text{read}(\text{chl}_e(\nu_c)))}\ \text{(chl recv)}$$

Both are blocking in case the targeted channels are full or empty respectively. Also both are mere thread-local transitions not effecting any other thread. To verify the state of a channel the operations *channel.full* and *channel.empty* are offered according to the rules

$$\frac{}{(e, \mathit{channel.full}(\nu_c)) \to_e (e, \text{full}(\text{chl}_e(\nu_c)))}\ \text{(chl full)}$$

and

$$\frac{}{(e, \mathit{channel.empty}(\nu_c)) \to_e (e, \text{empty}(\text{chl}_e(\nu_c)))}\ \text{(chl empty)}$$

This completes the set of inference rules defining valid state transitions utilized to formalize the semantic of our IR.

Example 3.20 (channel operations)**.** The utilization of channels always involves a variety of statements and instructions such that even for small examples the transition-relation based notation utilized to formalize their semantic is impractical for providing any meaningful example. Instead we provide a example including channel operations and describe the allowed transitions utilizing a formalism focusing on the most relevant parts regarding channels. The basis of our example is given by the code fragment

```
auto c = channel.create(int<4>,2);
merge(spawn(job[2,2]()⇒ {
  for(int<4> i = 0 .. 5 : 1 ) {
    if (getThreadID(0) == 0) {
      channel.send(c, produce());
    } else {
      consume(channel.recv(c));
    }
  }
}));
```

including three threads. The main thread creates a channel c, spawns two threads and waits for their completion. The two inner threads implement a simple producer/consumer pattern based on channel c. The first thread produces values and sends them to the channel while the second thread retrieves and consumes those values in order.

To model the execution state of this example code we utilize a triple

$$(c_b, p_m, p_1, p_2)$$

where $c_b \in \mathbb{N}^*$ is a tuple describing the state of the buffer utilized by the channel c, $p_m \in \mathbb{N} \cup \{\times\}$ the program point of the main thread based on referencing a line in the given code fragment, and $p_1, p_2 \in \mathbb{N} \cup \{\times\}$ the iteration number currently processed by the two inner threads respectively.

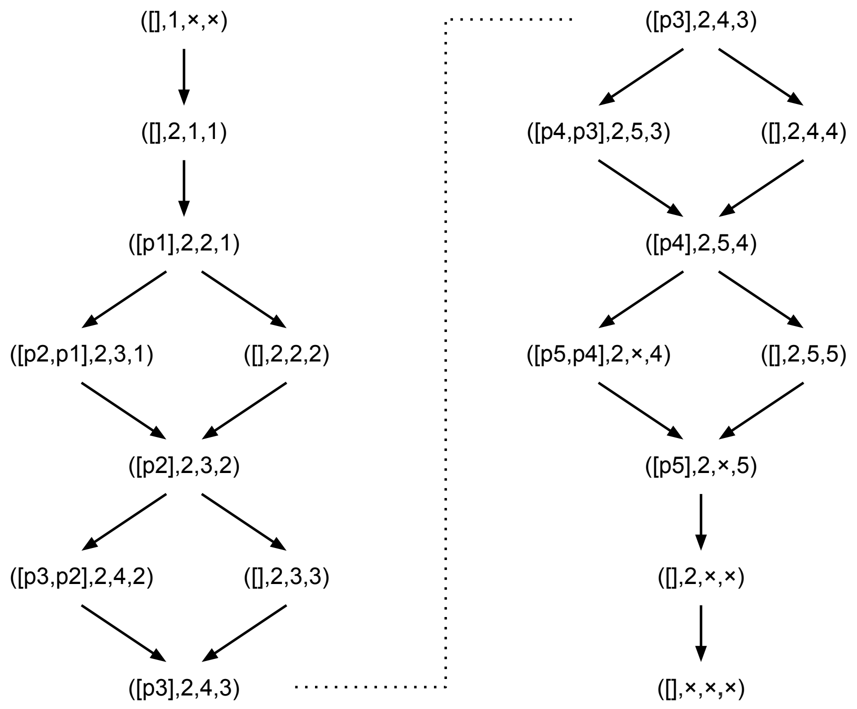


Figure 3.5: Reachable States of Example 3.20

The program state $\times$ indicates that the corresponding thread is not running at all. Hence, e.g. $([p1, p3], 2, 3, 1)$ would correspond to a state where the channel buffer contains the products $p1$ and $p3$, the main thread is processing line 2, the first inner thread is about to process line 5 of its 3rd iteration and the second inner thread is about to process line 7 of its 1st iteration. Note that this representation is a abstract summary of the information covered by the states connected by the global transition relation $\rightarrow$.

The state transitions supported by this setup are illustrated in Figure 3.5. Each of the shown transitions is based on an element of the $\rightarrow^*$ relation. The represented program starts by spawning two threads which produce and consume data, thereby never exceeding the boundaries of the channel utilized for the data exchange. Send operations are blocked whenever the channel is full and receive operations whenever the channel is empty. For instance, there is no transition from state $([p2, p1], 2, 3, 1)$ to the invalid state $([p3, p2, p1], 2, 4, 1)$ since the capacity of the utilized channel is 2. Also, there is no transition from state $([], 2, 4, 4)$ to the valid, yet not reachable state $([], 2, 4, 5)$ since the receive operation processed by the second thread is blocked until some data is available in the channel buffer.

3.8 Extensions

After formalizing the IR constructs, restricting their composition to form valid statements and the formalization of the semantic of type, expression and statement constructs, this section focuses on language extensions built on top of the infrastructure established by the language core. The section starts by summarizing the means offered for defining language extensions, which is then followed by descriptions of important instances.

3.8.1 Extension Mechanisms

Language extensions consist of the introduction of types, constants and operators. Types and operators might either be *abstract*, hence their actual internal implementation remains undefined, or *concrete* by providing a definition combining predefined types, operators and language constructs offered by the core or other extensions.

Types

Means Extensions may introduce *parametrized abstract types* by utilizing the abstract type construct or by composing new types utilizing the supported type constructors (see Definition 3.1). In case of the introduction of a closed abstract type $a \in \mathbb{T}_c$ its associated domain D_a has to be defined while for composed types constructed by combining existing types the domains are fixed by Definition 3.25.

Sub-Typing In addition to the domains, sub-type relations between introduced types may be added according to Definition 3.27. The utilization of sub-type relations can significantly reduce the number of operator overloads required for modeling operations based on new types.

Constants

Means Each extension may define constant values for their introduced, closed types. For instance, the extension handling boolean values is introducing two constants representing the truth values *true* and *false*. This kind of constants are integrated into INSPIRE by utilizing literal expressions exhibiting a corresponding identifier and type.

For a more convenient interaction with constants, extensions may define syntactic sugar for the concrete syntax of INSPIRE backed up by constructs of the abstract syntax. As for the core constructs, the impact of this abbreviations is limited to the text-based representation of INSPIRE code fragments.

Semantic To integrate constants defined by extensions into the semantic specification of INSPIRE, interpretations for literals have to be provided. The corresponding interface has been defined in Section 3.7.1. Essentially an interpretation $I \in \mathcal{I}$ has to be provides such that $\text{dom}(I)$ is a super-set of the introduced constants and every constant c of type t is mapped to an element $I[c] \in D_t$, hence, to an element of the corresponding domain. The resolution of the corresponding value is covered by the inference rule (lit) introduced on page 98.

Operators

Means Like types, operators can be introduced by extensions as an *abstract* construct utilizing a literal expression of a corresponding function type or as a *concrete* element by providing a lambda-expression implementing its functionality. The former variant provides the possibility of introducing black-box operations abstracting from operations conducted on abstract types while the latter is a transparent approach that can be inspected by e.g. program analysis.

Abstract operators are utilized if the internal representation should be shielded from the details of an operation. For instance, the exact details of summing up two arithmetic values on a bit level is beyond the scope of INSPIRE and hence this operation will be realized as a abstract black-box operation. Another use case are operators manipulating data structures like linked lists, sets or trees. The details of those operations may be excluded from the representation. Instead they are explicitly considered by analysis understanding their semantic implications. In general, when analyzing or processing code fragments including abstract operators, those either have to be treated explicitly or by applying the most conservative assumptions.

Introducing operators by providing an IR based implementation, on the other hand, opens them up for being processed by preexisting utilities. Analysis and transformations understanding the elements and constructs employed for defining *concrete* operators can derive the effects of those directly from their implementation. Here we utilize the fact that an actual implementation of an operator is its most accurate semantic description. Since the interpretation of *concrete* operators is derived from their definitions they are also referred to as *derived operators.*

For both approaches, *abstract* or *derived* operators, semantic sugar may be defined to make representations more easily digestible for humans.

Validity Furthermore, the utilization of operators introduced by extensions may be constraint by extending the rules of Definition 3.37 and Definition 3.38 regarding the structure of valid expressions and statements.

Semantic The integration of *derived* operators into the semantic framework established in Section 3.7 is covered by the provided IR-based implementation. A *derived* operator is equivalent to its implementation and can hence be substituted by it.

The semantic of an *abstract* operator, on the other hand, has to be defined similar to constants. An interpretation $I \in \mathcal{I}$ has to be provided by the extension introducing an abstract operator o of type $(t_1, \ldots, t_n) \to t$ such that $I[o]$ is a function

$$(\mathcal{E} \times D(t_1) \times \ldots \times D(t_n)) \to (\mathcal{E} \times D(t))$$

accepting an environment and a list of argument values and returning a potentially modified environment and the computed result. This interpretation is integrated into the small-step semantic transitions of Section 3.7 by the inference rules (lit) and (call value).

For the frequent case of pure interpretation functions not effecting the environment we consider a interpretation function

$$f : (D(t_1) \times \ldots \times D(t_n)) \to D(t)$$

equivalent to the function

$$f' : (\mathcal{E} \times D(t_1) \times \ldots \times D(t_n)) \to (\mathcal{E} \times D(t))$$

defined by $f'(e, d_1, \ldots, d_n) = (e, f(d_1, \ldots, d_n))$ throughout this section. This convention eliminates the requirement of forwarding unaffected global environments when handling pure functions.

3.8.2 Important Extensions

In the following a set of modular extensions are covered. A few of those have already been mentioned and utilized within previous sections. At this point a complete specification of those is provided.

Meta Types

In several cases operators depending on types are required. For instance, the operator *channel.create* requires an argument determining the type of value to be transferred through the resulting channel and another fixing the size of the buffer to be allocated.

Types The bridge between types and values within the IR is formed by the generic meta type family

$$type\,\langle t \rangle$$

where the type $t \in \mathbb{T}$ is the type to be represented. The domain $D_{type\langle t \rangle}$ is defined by the singleton set $\{t\} \subset \mathbb{T}$. Consequently, every expression of the given type is always evaluated to the same value. This value is represented by the literal

$$lit(\text{'t'} : type\,\langle t\rangle)$$

where 't' $\in \mathbb{I}$ is a string-representation of the represented type. Hence, $I[lit(\text{'t'} : type\,\langle t\rangle)] = t$ for all $t \in \mathbb{T}$. Consequently, the literal

$$lit(\text{'A'} : type\,\langle A\rangle)$$

is representing the generic type A and

$$lit(\text{'}struct\{x : B\,\langle A\rangle\}\text{'} : type\,\langle struct\{x : B\,\langle A\rangle\}\rangle)$$

a struct type containing a single member x of the parametrized generic type $B\,\langle A\rangle$. As for other literals we will omit the extra syntax and will only write 'A' or A instead of $lit(\text{'A'} : type\,\langle A\rangle)$ if its interpretation is clear from the context.

Numerical Meta Types In several cases static numerical values need to be incorporated into types. For instance the capacity of a channel or the size of vector of elements should be properly reflected within the corresponding type. This is realized by the type family

$$i'\,\langle\rangle$$

where $i' \in \mathbb{I}$ is the string-representation of an integer $i \in \mathbb{N}_0$. The associated domain is defined by $D_{i'} = \emptyset$. For instance, $4 = 4\,\langle\rangle$ is a numerical meta type representing the integer value 4. Its domain is given by $D_4 = \emptyset$. Hence, numerical meta types are pure meta types for modeling concepts in the type system. No expression can exhibit a numerical meta type since non could be evaluated to a value of their domains – since those are empty.

Numerical Parameter Meta Types To utilize numerical meta types the type family

$$param\,\langle t\rangle$$

where $t = i'\,\langle\rangle$ for some $i \in \mathbb{N}_0$ is defined. The associated domain is given by $D_{param\langle t\rangle} = \{t\}$ similar to the meta type family $type\,\langle\ldots\rangle$. Literals of *parameter* meta types have the shape

$$lit(i' : param\,\langle i'\,\langle\rangle\rangle)$$

which we identify with $lit(i' : param\,\langle i'\rangle)$ and i' if its interpretation is clear from the context. The interpretation of those literals is given by

$$I[lit(i' : param\,\langle i'\,\langle\rangle\rangle))] = i'\,\langle\rangle \in \mathbb{T}$$

for all $i \in \mathbb{N}_0$. Hence, the *param* type family is a variant of the *type* family restricted to numerical meta types.

Primitive Types

The next class of extensions covers primitive types which already have been encountered in the syntactic definition of IR expressions and statements.

Unit The $unit\left\langle\right\rangle$ type is a generic type to be assigned to operations causing side effects but not omitting actual results. For instance the *channel.send* of type

$$(channel\left\langle\alpha,\beta\right\rangle,\alpha)\rightarrow unit$$

is one of those operations. While not obtaining any information it causes the desirable side-effect of submitting a value to a given channel.

The domain $D_{unit\langle\rangle}$ of the $unit\left\langle\right\rangle$ type is given by $D_{unit\langle\rangle}=\{unit\}$ where the element *unit* is a token not present in any other domain. The literal $lit(unit : unit\left\langle\right\rangle)$ is representing this value and its interpretation is fixed to $I[lit(unit : unit\left\langle\right\rangle)] = unit$.

Boolean The generic type $bool\left\langle\right\rangle$ is utilized for representing the set of truth values. Its domain is given by $D_{bool}=\mathbb{B}=\{true, false\}$. Furthermore the two constants

$$lit(true : bool)$$

and

$$lit(false : bool)$$

with their interpretation $I[lit(true : bool)] = true$ and $I[lit(false : bool)] = false$ are defined by this extension.

On top of the *bool* type and its constants several operators are defined. All of them are derived operators implemented by composing predefined constructs and primitives, as listed in Table 3.1.

Since the listed boolean operators are defined as derived operators their semantic can be deduced from their definitions. For instance, an analysis embedded in a framework supporting the semantic of the core language constructs and their composition could be capable of deducing that *bool.neg* is a pure function and the result of $bool.neg(true)$ is $false$. Furthermore any future analysis may inspect the provided definitions to obtain required data regarding those operators. No fact-database covering effects of these operators has to be established nor maintained.

However, backends converting IR code into target code (e.g. C) are not actually synthesize code containing functions implementing the listed operators. Instead applications of those operators are recognized and converted to the actual built-in operators offered by target languages.

Name	Type	Definition
bool.neg	$(bool) \to bool$	`(bool a) { if (a) return false; return true; }`
bool.and	$(bool, bool) \to bool$	`(bool a, bool b) { if (a) return b; return false; }`
bool.or	$(bool, bool) \to bool$	`(bool a, bool b) { if (a) return true; return b; }`
bool.eq	$(bool, bool) \to bool$	`(bool a, bool b) { if (a) return b; return bool.neg(b); }`
bool.ne	$(bool, bool) \to bool$	`(bool a, bool b) { return bool.neg(bool.eq(a,b)); }`

Table 3.1: List of (derived) boolean operators.

Although common in logic the definition of the conjunction operator above is not widely utilized within actual programming languages. In real world languages *short-circuit evaluation* for boolean expression is the standard and their proper representation is crucial for the correct representation of a majority of input codes.

Since all operator applications within our IR are represented by function calls to keep the number of language constructs low, all arguments are always evaluated before the actual function is called (call-by-value semantic). However, *short-circuit evaluation* demands that some parameters are only evaluated under certain conditions. For instance in the C code fragment

```
if (f() && g()) { ... }
```

the function *g* is only evaluated in case *f()* evaluates to *false*. To properly represent this behavior the operators of Table 3.2 are utilized within INSPIRE. Unlike the previous definitions, the *lazy* alternatives accept the second argument as a lazy evaluated boolean expression of the closure type $() \Rightarrow bool$ which will only be processed in case the first argument is not sufficient to determine the value of the boolean operator.

The example above would therefore be encoded within INSPIRE by

```
if (bool.land(f(), ()⇒g())) { ... }
```

Name	Type	Definition

bool.land	$(bool, () \Rightarrow bool) \rightarrow bool$	`(bool a, ()⇒bool b) {`
		`   if (a) return b();`
		`   return false;`
		`}`

bool.lor	$(bool, () \Rightarrow bool) \rightarrow bool$	`(bool a, ()⇒bool b) {`
		`   if (a) return true;`
		`   return b();`
		`}`

Table 3.2: Short-circuit evaluated, derived boolean operators.

where ()⇒g() is a bind expression wrapping up the evaluation of g into a lazy expression of type $() \Rightarrow bool$. Finally, the last boolean-related operator is the trinary *if-then-else* operator which is covered by the generic *bool.ite* operator as follows:

Name	Type	Definition

bool.ite	$(bool, () \Rightarrow \alpha, () \Rightarrow \alpha) \rightarrow \alpha$	`(bool a, ()⇒α b,()⇒α c) {`
		`   if (a) return b();`
		`   return c();`
		`}`

Here the resulting value is the value computed by one of the two passed lazy expression b and c. The decision which of the two has to be evaluated depends on the value of a.

Since the full notation of the definitions nor the abbreviated names of the boolean operators are neither convenient for being handled by humans we will utilize the familiar C operators, their infix notation, *short-circuit evaluation*, parenthesis and precedence order within the *concrete* INSPIRE syntax. For instance, by utilizing this convention the concrete syntax based code fragment

```
(bool a, bool b) {
   return !(a == b);
}
```

is (structurally) equivalent to the definition of the *bool.ne* operator provided above.

Arithmetic For arithmetic operations in C like languages three cases have to be distinguished – operations on signed integers, unsigned integers and floating point values. Further, for every case, the precision has to be determined. Within INSPIRE the three type families

$$int\langle t_1\rangle\,,\ uint\langle t_1\rangle\,,\ \text{and}\ real\langle t_2\rangle$$

are utilized to distinguish those three cases where $t_1 \in \{i\langle\rangle \in \mathbb{T} \mid i \in \{1,2,4,8\}\}$ and $t_2 \in \{i\langle\rangle \in \mathbb{T} \mid i \in \{4,8\}\}$ are numerical type parameters. Those parameters determine the precision of the represented arithmetic type. For instance, $\mathit{int}\langle 4\rangle$ represents a signed 4-byte integer encoded using two's complement and $\mathit{uint}\langle 2\rangle$ a unsigned 2-byte integer. The types $\mathit{real}\langle 4\rangle$ and $\mathit{real}\langle 8\rangle$ represent the single and double precision floating point formats defined by IEEE 754-2008 standard. The domains of the types are defined accordingly. Outlining all the associated details is straight forward, yet clearly beyond the scope of this thesis.

Due to their value domains some numerical types are sub-types of others. This relation is covered by the inference rules

$$\frac{a \leq b}{\mathit{int}\langle a\rangle <: \mathit{int}\langle b\rangle}\text{(int)}$$

$$\frac{a \leq b}{\mathit{uint}\langle a\rangle <: \mathit{uint}\langle b\rangle}\text{(uint)}$$

$$\frac{a < b}{\mathit{uint}\langle a\rangle <: \mathit{int}\langle b\rangle}\text{(u2s)}$$

$$\frac{}{\mathit{real}\langle 4\rangle <: \mathit{real}\langle 8\rangle}\text{(real)}$$

extending the sub-type relations started in Section 3.27. Within the same type family a type with a smaller precision is always a sub-type of types with larger precision. Additionally an unsigned integer type is a sub-type of a signed integer type whenever its precision is strictly smaller than the super-types precision.

Numerical constants can be introduced by corresponding literals. For instance the literal

$$\mathit{lit}(8 : \mathit{int}\langle 2\rangle)$$

represents the 2-byte signed integer 8 and

$$\mathit{lit}(-2.4 : \mathit{real}\langle 8\rangle)$$

the double value -2.4. The interpretation I is extended accordingly

Unlike for the boolean type, arithmetic operators are incorporated by abstract literals not exhibiting IR based definitions. The following table provides a summery of the basic operators.

Name	Type	Description
$int.add$	$(int\langle\alpha\rangle, int\langle\alpha\rangle) \to int\langle\alpha\rangle$	Signed Addition
$int.sub$	$(int\langle\alpha\rangle, int\langle\alpha\rangle) \to int\langle\alpha\rangle$	Signed Subtraction
$int.mul$	$(int\langle\alpha\rangle, int\langle\alpha\rangle) \to int\langle\alpha\rangle$	Signed Multiplication
$int.div$	$(int\langle\alpha\rangle, int\langle\alpha\rangle) \to int\langle\alpha\rangle$	Signed Division
$int.mod$	$(int\langle\alpha\rangle, int\langle\alpha\rangle) \to int\langle\alpha\rangle$	Signed Modulo
$uint.add$	$(uint\langle\alpha\rangle, uint\langle\alpha\rangle) \to uint\langle\alpha\rangle$	Unsigned Addition
$uint.sub$	$(uint\langle\alpha\rangle, uint\langle\alpha\rangle) \to uint\langle\alpha\rangle$	Unsigned Subtraction
$uint.mul$	$(uint\langle\alpha\rangle, uint\langle\alpha\rangle) \to uint\langle\alpha\rangle$	Unsigned Multiplication
$uint.div$	$(uint\langle\alpha\rangle, uint\langle\alpha\rangle) \to uint\langle\alpha\rangle$	Unsigned Division
$uint.mod$	$(uint\langle\alpha\rangle, uint\langle\alpha\rangle) \to uint\langle\alpha\rangle$	Unsigned Modulo
$real.add$	$(real\langle\alpha\rangle, real\langle\alpha\rangle) \to real\langle\alpha\rangle$	FP Addition
$real.sub$	$(real\langle\alpha\rangle, real\langle\alpha\rangle) \to real\langle\alpha\rangle$	FP Subtraction
$real.mul$	$(real\langle\alpha\rangle, real\langle\alpha\rangle) \to real\langle\alpha\rangle$	FP Multiplication
$real.div$	$(real\langle\alpha\rangle, real\langle\alpha\rangle) \to real\langle\alpha\rangle$	FP Division

Furthermore comparison operators $(<, \leq, =, \neq, \geq, >)$, conversion operators within the same type family like

$$int.to.int : (int\langle\alpha\rangle, param\langle\beta\rangle) \to int\langle\beta\rangle$$

and and between families like

$$real.to.int : (real\langle\alpha\rangle, param\langle\beta\rangle) \to int\langle\beta\rangle$$

are covered. Those conversion functions demonstrate another application of the numerical parameter meta type *param*.

All of the operators are interpreted by pure functions exhibiting the expected behavior. For instance, $I[int.add]$ is given by the function

$$f : (D_{int\langle\alpha\rangle} \times D_{int\langle\alpha\rangle}) \to D_{int\langle\alpha\rangle}$$

defined by $(x, y) \mapsto x +_\alpha y$ where $+_\alpha$ is the addition operator of the signed α-byte integers.

As for the boolean operators the operator symbols, overloads, precedence order, literal syntax and infix notation of the C language family is imported in our concrete syntax formulation to avoid the explicit notation of the actual IR constructs.

Character The type families *char* (8-bit) and *wchar* (16-bit), character literals, char-based comparison operators and conversion functions from and to the integral types are provided by the character extension. As for the arithmetic package all of those are abstract and their detailed specification is exceeding the scope of this section. Syntactically, as within the C language family, literals of the shape `'c'` are utilized to represent character values. However, there is no implicit conversion between characters and numeric values or booleans – explicit conversions need to be applied.

Containers

In addition to primitive (scalar) types constructs for composed data structures are required to modeling more advanced language features.

Lists The type family $list\,\langle t\rangle$ represents a sequence of ordered elements of type t. The associated domains are defined by

$$D_{list\langle t\rangle} = D_t^*$$

for all $t \in \mathbb{T}$, hence, the set of all sequences of elements of the domain D_t. Furthermore the two abstract constructors

$$empty : (type\,\langle\alpha\rangle) \to list\,\langle\alpha\rangle$$

and

$$cons : (\alpha, list\,\langle\alpha\rangle) \to list\,\langle\alpha\rangle$$

are offered for assembling lists. Their interpretation is given by $I[empty] = f_{empty}$ and $I[cons] = f_{cons}$ where

$$f_{empty}(x) = []$$

for all $x \in \mathbb{T}$ and

$$f_{cons}(x, [y_1, \ldots, y_n]) = [x, y_1, \ldots, y_n]$$

for all $x, y_1, \ldots, y_n \in \mathcal{V}$.

In the *concrete syntax* the construct [] is utilized to represent a call to the *empty* constructor with a proper type (if it is clear from the context) and the construct $[x_1, \ldots, x_n]$ is equivalent to the term

$$cons(x_1, cons(x_2, cons(\ldots cons(x_n, []) \ldots)))$$

gradually assembling the corresponding sequence.

Lists are mainly utilized for initializing other homogeneous collections of elements like arrays or vectors which are covered next.

Arrays Members of the type family $\mathit{array}\left\langle\alpha\right\rangle$ represent homogeneous sequences of ordered elements whose length is dynamically determined at their creation point. As for lists the domain of an array is determined by

$$D_{\mathit{array}\left\langle t\right\rangle} = D_t^*$$

where D_t is the domain of the element type t. The operators

$$\mathit{array.create} : (\mathit{type}\left\langle\alpha\right\rangle, \mathit{uint}\left\langle 8\right\rangle) \to \mathit{array}\left\langle\alpha\right\rangle$$

and

$$\mathit{array.create} : (\mathit{list}\left\langle\alpha\right\rangle, \mathit{uint}\left\langle 8\right\rangle) \to \mathit{array}\left\langle\alpha\right\rangle$$

can be utilized to create array values containing undefined elements or the elements of the given list respectively. In both cases the second parameter determines the length of the resulting array. Furthermore the subscript operator

$$\mathit{array.subscript} : (\mathit{array}\left\langle\alpha\right\rangle, \mathit{int}\left\langle 8\right\rangle) \to \alpha$$

can be utilized to extract an element from an array by specifying its index within the represented sequence. The interpretation of those abstract symbols is fixed accordingly by

$$I[\mathit{array.create}](t, s) = [v_0, \ldots, v_{s-1}]$$

where $v_0, \ldots, v_{s-1}$ are arbitrary elements of D_t,

$$I[\mathit{array.create}]([v_0, \ldots, v_n], s) = [v_0, \ldots, v_{s-1}]$$

where for all $n < i < s$ the value v_i is an arbitrary element of D_t and

$$I[\mathit{array.subscript}]([v_0, \ldots, v_n], i) = \begin{cases} v_i & \text{if } 0 \leq i < n \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

realizing the projection to a component of the represented value.

Within the concrete syntax the *array.subscript* operator is abbreviated by $a[i]$ where a is the expression computing the array to be accessed and i the index of the requested element.

Vectors For homogeneous sequences of elements with a statically fixed length the type family $\mathit{vector}\left\langle t, s\right\rangle$ is defined where $t \in \mathbb{T}$ is the element type to be stored within the sequence and the numerical type parameter $s \in \{i\left\langle\right\rangle \in \mathbb{T} \mid i \in \mathbb{N}_0\}$ the length. The domains are defined by

$$D_{\mathit{vector}\left\langle t,s\right\rangle} = D_t^s$$

As for arrays the two constructors

$$\mathit{vector.create} : (\mathit{type}\left\langle\alpha\right\rangle, \mathit{param}\left\langle\beta\right\rangle) \to \mathit{vector}\left\langle\alpha, \beta\right\rangle$$

and

$$vector.create : (list\langle\alpha\rangle, param\langle\beta\rangle) \rightarrow vector\langle\alpha,\beta\rangle$$

are offered. Unlike for arrays the size of the resulting vector has to be statically fixed by a numerical type parameter instead of a dynamically evaluated integer expression. To access an element contained within a vector the array subscript operator can be utilized since vector types are sub-types of the corresponding array types. This relation is incorporated by the inference rule

$$\frac{}{vector\langle t,s\rangle <: array\langle t\rangle}\ (\text{a2v})$$

Additionally generic manipulation operations like

$$vector.reduce : (vector\langle\alpha,\beta\rangle, \gamma, (\beta,\gamma) \rightarrow \gamma) \rightarrow \gamma$$

and higher order operators, including the pointwise operator

$$vector.pointwise : ((\alpha) \rightarrow \beta) \rightarrow ((vector\langle\alpha,\gamma\rangle) \rightarrow vector\langle\beta,\gamma\rangle)$$

converting an operator of type $(\alpha) \rightarrow \beta$ into a vectorized variation, are covered for modeling SIMD instructions[15].

Mutable State

So far all the primitives and constructs introduced by the language core and the covered extensions are restricted to handling immutable data objects. For instance, we can create vectors of four integers and we can implement functions creating copies of those vectors where selected components are substituted by alternative values but we do not have the ability to alter the content of a given vector instance. This characteristic is typical for functional languages and does not limit expressibility. However, since INSPIRE is intended to cover imperative languages, which are based on the fundamental concept of applying sequences of operations on mutable memory locations, corresponding support for mutable data is required.

The approach we have taken for modeling mutable memory locations is orthogonal to the remaining extensions and the language core itself. Hence, details may be altered without effecting any other extension or the core and on the other hand future extensions can fully utilize the constructs of the mutable state extension.

[15]SIMD = single instruction, multiple data; utilized for exploiting instruction level parallelism using vectorization

Syntax A mutable memory location is addressed by a value of the type family

$$ref\,\langle t\rangle$$

where $t \in \mathbb{T}$ determines the type of value stored within the referenced location. Its content may be retrieved utilizing the abstract function

$$ref.deref : (ref\,\langle\alpha\rangle) \to \alpha$$

and updated by the abstract operator

$$ref.assign : (ref\,\langle\alpha\rangle\,,\alpha) \to unit$$

For the concrete syntax we will utilize the familiar unary $*$ operator in prefix notation for the application of the $ref.deref$ operator or consider it an implicit operation exhibiting no explicit notation at all. For instance, let x be a variable of type $ref\,\langle t\rangle$ for some $t \in \mathbb{T}$. Then the expressions $ref.deref(x)$ and $*x$ are equivalent. Further, the concrete syntax terms $*x$ and x are considered equivalent if the deref operation can be deduced from the context.

The assignment operator, on the other hand, is represented by the binary assignment operators $=$ in in-fix notation when utilizing the concrete syntax. In special cases, where the distinction between the initialization of a variable, which is also utilizing the $=$ symbol, and an assignment should be stressed, the more accurate $:=$ assignment operator is employed. As for other operators we will inherit operator precedences from the C language family.

Creation and Destruction To create a fresh memory location the operator

$$ref.alloc : (type\,\langle\alpha\rangle\,, memloc) \to ref\,\langle\alpha\rangle$$

is provided. Its application is allocating a memory location suitable to maintain an instance of the type specified by the first parameter and returns a value referencing it. The second parameter of type *memloc* determines whether the location should be allocated on the *heap*, the *stack* or other address spaces like potential scratchpad memory within embedded systems or some kind of device memory available on OpenCL devices. For each case constants like

$$memloc.stack : memloc$$

are offered.

Beside specifying the targeted memory segment, the selection of the memory location also determines the life cycle of the allocated location. For instance, while stack-allocated locations will be automatically freed at the

Name	Type	Definition
$ref.var$	$(type\,\langle\alpha\rangle) \rightarrow ref\,\langle\alpha\rangle$	(type ⟨α⟩ t) { **return** ref.alloc(α, memloc.stack); }
$ref.var$	$(\alpha) \rightarrow ref\,\langle\alpha\rangle$	(α v) { **auto** r = ref.var(α); r = v; **return** r; }
$ref.new$	$(type\,\langle\alpha\rangle) \rightarrow ref\,\langle\alpha\rangle$	(type ⟨α⟩ t) { **return** ref.alloc(α, memloc.heap); }
$ref.new$	$(\alpha) \rightarrow ref\,\langle\alpha\rangle$	(α v) { **auto** r = ref.new(α); r = v; **return** r; }

Table 3.3: Derived memory allocation operators.

end of their surrounding scopes, heap allocated locations are required to be released explicitly by utilizing the operator

$$ref.delete : (ref\,\langle\alpha\rangle) \rightarrow unit$$

Frequently memory locations need to be allocated containing some undefined value or allocated and initialized with some value within a single expression. For this cases we offer the overloaded, derived operators listed in Table 3.3 which are abbreviate in the concrete syntax utilizing the keywords *var* and *new*.

Comparison Operators The operator

$$ref.eq : (ref\,\langle\alpha\rangle\,, ref\,\langle\beta\rangle) \rightarrow bool$$

determines whether two references are addressing the same memory location. Correspondingly $ref.ne$ is a derived operator negating the result of $ref.eq$. As usual we will utilize the C operators == and != when comparing references.

Null Reference To represent the value of a reference not referencing any memory location (e.g. since it has not yet been initialized) the constant

$$ref.null : ref\,\langle none\rangle$$

is utilized where the generic abstract type *none* is an abstract type with an empty domain ($D_{none} = \emptyset$). To check whether a given reference is a null reference the derived operator $ref.is.null$ defined by

$$(ref{<}\alpha{>}\ r) \to bool\ \{\ \textbf{return}\ r == ref.null;\}$$

is utilized.

Sub-References Typically the data stored within memory location is structured. For instance, when storing a struct composed of several fields within a memory location, fractions of the memory location are utilized for the various fields. By sub-referencing we are referring to the support of obtaining references to the nested fields of structured data within memory locations (and the reverse process). It is based on the abstract operator

$$ref.narrow : (ref\,\langle\alpha\rangle\,, datapath, type\,\langle\beta\rangle) \to ref\,\langle\beta\rangle$$

and its inverse operator

$$ref.expand : (ref\,\langle\alpha\rangle\,, datapath, type\,\langle\beta\rangle) \to ref\,\langle\beta\rangle$$

where the *datapath* parameter addresses a sub-structure within a structured memory location. Corresponding values can be constructed using the constant

$$dp.root : datapath$$

and three constructors – one addressing fields within structs

$$dp.member : (datapath, identifier) \to datapath$$

a second unpacking values from a union value

$$dp.unpack : (datapath, type\,\langle\alpha\rangle) \to datapath$$

and a third addressing elements within arrays or vectors

$$dp.element : (datapath, int\,\langle 8\rangle) \to datapath$$

More data path constructors may be added by additional extensions to navigate data structures and containers introduced by those.

The constant *dp.root* is addressing a full structure and every application of a constructor is narrowing down the addressed object to a part of the parent structure. For instance, the value of the expression

$$dp.element(dp.root, 12)$$

is a path to the 12-th element of an array or vector.

The application of a $ref.narrow$ starts with a given reference, follows the provided data path and returns a reference to the element reached by the last step. Since the type of the resulting reference can not be deduced via the type system it has to be provided by an extra argument. The operator $ref.expand$ is the inverse operation navigating from a sub-reference to a reference addressing an enclosing structure.

Example 3.21 (narrow and expand)**.** Let v be a variable of type

$$ref\,\langle array\,\langle struct\{x : int\,\langle 4\rangle\,, x : int\,\langle 4\rangle\}\rangle\rangle$$

The expression

$$ref.narrow(v, dp.element(dp.root, 12), struct\{x : int\,\langle 4\rangle\,, y : int\,\langle 2\rangle\})$$

is of type $ref\,\langle struct\{x : int\,\langle 4\rangle\,, y : int\,\langle 2\rangle\}\rangle$ and referencing the 12-th element of the array referenced by v. Similar

$$ref.narrow(v, dp.member(dp.element(dp.root, 12), x), int\,\langle 4\rangle)$$

of type $ref\,\langle int\,\langle 4\rangle\rangle$ is referencing the x field of the 12-th element of the array referenced by v. Let f be the result of the last expression. Than

$$ref.expand(f, dp.member(dp.root, x), struct\{x : int\,\langle 4\rangle\,, y : int\,\langle 2\rangle\})$$

of type $ref\,\langle struct\{x : int\,\langle 4\rangle\,, y : int\,\langle 2\rangle\}\rangle$ is a reference to the struct where the reference f is targeting the field x.

Applications of the $ref.narrow$ and $ref.expand$ operator can be utilized to navigate freely within structured memory locations. However, for special cases derived operators and associated concrete syntax constructs are utilized. The four main derived operators are

Name	Definition
$array.ref.elem$	(ref $\langle array\,\langle\alpha\rangle\rangle$ a, int $\langle 8\rangle$ i) { **return** ref.narrow(a, dp.element(dp.root, i), α); }
$vector.ref.elem$	(ref $\langle vector\,\langle\alpha\rangle\rangle$ a, int $\langle 8\rangle$ i) { **return** ref.narrow(a, dp.element(dp.root, i), α); }
$struct.ref.elem$	(ref $\langle\alpha\rangle$ a, identifier i, type $\langle\beta\rangle$ t) { **return** ref.narrow(a, dp.member(dp.root, i), t); }
$union.ref.elem$	(ref $\langle\alpha\rangle$ a, type $\langle\beta\rangle$ t) { **return** ref.narrow(a, dp.unpack(dp.root, t), t); }

covering the access of array, vector and struct and union values. Note that a distinction between arrays and vectors is required since sub-typing rules between arrays and vectors are not extended to references of those. Also a generic type for the input references of the *struct.ref.elem* and *union.ref.elem* operators has to be utilized since no generic type restricting inputs to structs or units can be specified within our IR's type system[16].

To avoid the extensive notation of the abstract IR syntax the C subscript operator `x[i]` and structure reference operator `x.f` are utilized within the concrete syntax to denote the corresponding operation. Nevertheless, internally all these operations are mapped to applications of the *ref.narrow* operator. The *ref.expand* on the other hand is required to model C conversions of pointer-to-scalar values to pointer-to-array values and in particular within C++ to convert a reference to a base type to a reference of a derived type – however, details of those are beyond the scope of this section.

Semantic In addition to the syntax of the constructs provided for handling mutable states the semantic of the involved types and abstract operators has to be defined.

Definition 3.54 (reference domain)**.** Let $\mathcal{L}$ be an arbitrary set of memory locations such that $\eta \notin \mathcal{L}$ where η is the value of the *null location.* Further let $\mathcal{P}$ bet the set of all data paths generated by the grammar

$$\begin{aligned} p &::= u \mid d \\ u &::= \bot \mid i.u \mid f.u \mid t.u \\ d &::= \bot \mid d.i \mid d.f \mid d.t \end{aligned}$$

where p is the starting symbol, $i \in \mathbb{Z}$ is an arbitrary index, $f \in \mathbb{I}$ is a field name identifier and $t \in \mathbb{T}$ a type. The domain of the reference type $ref\,\langle t\rangle$ is defined by

$$D_{ref\langle t\rangle} = (\mathcal{L} \times \mathcal{P}) \cup \{\eta\} = \mathcal{R}$$

Hence, a reference is either *null* or a pair of a memory location and a data path addressing the targeted sub-structure within the associated location.

Note that the definition provides the possibility of forming paths in both directions – addressing sub-structures (e.g. $\bot.n.3$) and super-structures (e.g. $x.\bot$). The capability of addressing super-structures is required to provide an interpretation for the *ref.narrow* and *ref.expand* operators such that those are total and the inverse of each other.

Example 3.22 (reference values)**.** Let $l \in \mathcal{L}$ be a memory location. The pair $(l, \bot) \in \mathcal{L} \times \mathcal{P}$ is a reference addressing the full structure stored within

[16]Extending the type system for this use case would be possible but its complexity is hardly justified by the additional gain.

memory location l while $(l, \perp.4)$ is addressing the 4-th component of the tuple of values stored within l. A reference $(l, 2.\perp)$ would referencing the super-structure containing the value of l as its second sub-component. However, since l is the full structure allocated at this location such a reference to a super-structure must not be dereferenced.

For the mutable state extension interpretations of the abstract constant $ref.null$ and the operators $ref.alloc$, $ref.delete$, $ref.eq$, $ref.narrow$, $ref.expand$, $ref.deref$, and $ref.assign$ have to be provided.

Null and the Equals Operator For the null-constant the interpretation is given by

$$I[ref.null] = \eta$$

where η is the *null location*. Also, the interpretation of the $ref.eq$ can be defined in a straightforward way by

$$I[ref.eq](a, b) = (a = b)$$

which is simply mapping the interpretation of the equality operator between references to the equality of elements of the set of references $\mathcal{R}$.

Creation and Destruction Let state : $\mathcal{L} \rightharpoonup \mathcal{V}$ be a partial mapping assigning memory locations the value stored within those. The evolution of this mapping during the course of the execution of a program fragment is the main focus of the mutable state extension. It is therefore maintained within the universal environment of the program state (see Definition 3.43).

The creation of a memory location is conducted by the $ref.alloc$ operator whose interpretation is given by

$$I[ref.alloc](e, t, m) = (\text{state}_e(l) = [], (l, \perp))$$

where $l \in \mathcal{L} \setminus \text{dom}(\text{state}_e)$ is a fresh location and $\perp$ the empty data path addressing the root structure. The new location is created and initialized with the default value $[] \in \mathcal{V}$ within the environment e.

Note that at this point we ignore the memory location designator m. A proper handling would require a formalization of the corresponding scopes and a partitioning of the set of memory locations $\mathcal{L}$ according to the utilized designators. Since this distinction is not required for the rest of the thesis we omit the associated details for brevity. Also, the handling of life cycles of memory locations is frequently omitted for similar reasons within related literature [83].

The delete operator is simply eliminating the mapping of a memory location from the environment. This is realized by its interpretation

$$I[ref.delete](e, (l, \perp)) = (\text{state}_e(l) \setminus l, unit)$$

which removing the value l from the domain of state$_e$. Note that the delete operator is not defined for references addressing sub-structures. The data path is required to be $\bot$. Consequently free operations may not be applied on sub-structures.

Data Paths Before we can specify the formalization of the remaining reference operators, data path constants and constructors have to be covered. Their interpretation is given by

$$I[dp.root]() = \bot$$

$$I[dp.member](p, f) = p.f$$

$$I[dp.unpack](p, t) = p.t$$

$$I[dp.element](p, i) = p.i$$

All of them produce a value of the set of data paths $\mathcal{P}$ according to their input arguments. We further define a utility function $|\cdot| : \mathcal{P} \to \mathbb{N}$ determining the length of a data path by

$$|a| = \begin{cases} 0 & \text{if } a = \bot \\ |a'| + 1 & \text{if } a = a'.x \\ |a'| - 1 & \text{if } a = x.a' \end{cases}$$

the function inv : $\mathcal{P} \to \mathcal{P}$ inverting a data path by

$$\text{inv}(a) = \begin{cases} \bot & \text{if } a = \bot \\ x.\,\text{inv}(a') & \text{if } a = a'.x \\ \text{inv}(a').x & \text{if } a = x.a' \end{cases}$$

the function head : $\mathcal{P} \to (\mathbb{N} \cup \mathbb{I} \cup \mathbb{T})$ obtaining the first element of a data path p where $|p| > 0$ by

$$\text{head}(p) = \begin{cases} x_1 & \text{if } p = \bot.x_1.x_2 \ldots x_n \\ \text{undefined} & \text{otherwise} \end{cases}$$

the function tail : $\mathcal{P} \to \mathcal{P}$ obtaining the remaining data path when eliminating the head element of a path p where $|p| > 0$ by

$$\text{tail}(p) = \begin{cases} \bot.x_2 \ldots x_n & \text{if } p = \bot.x_1.x_2 \ldots x_n \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the function concat : $\mathcal{P} \times \mathcal{P} \to \mathcal{P}$ computing the concatenation of two data paths a and b by

$$\text{concat}(a,b) = \begin{cases} a & \text{if } b = \bot \\ \text{concat}(a, b').x & \text{if } |a| \geq 0 \text{ and } b = b'.x \\ \text{concat}(a', \text{tail}(b)) & \text{if } a = x.a' \text{ and } \text{head}(b) = x \\ \text{undefined} & \text{if } a = x.a' \text{ and } \text{head}(b) \neq x \\ \text{inv}(\text{concat}(\text{inv}(a), \text{inv}(b))) & \textit{otherwise} \end{cases}$$

Example 3.23 (data path handling)**.** Let $a = \bot.1.x.2$ be a data path, then $\text{len}(a) = 3$, $\text{inv}(a) = 2.x.1.\bot$, $\text{head}(a) = 1$ and $\text{tail}(a) = \bot.x.2$. If two data paths both have positive or both negative length the concatenation of those two paths is the simple concatenation of the involved steps. For instance,

$$\text{concat}(\bot.1.2, \bot.x) = \bot.1.2.x$$

and

$$\text{concat}(x.\bot, y.\bot) = y.x.\bot$$

If the length of the two data paths is not exhibiting the same sign the first path is consumed by the second. For instance,

$$\text{concat}(a.3.f.\bot, \bot.a.3) = f.\bot$$

and

$$\text{concat}(\bot.c.0, 0.\bot) = \bot.c$$

If the consumption is crossing the root path $\bot$ both temporary paths are pointing in the same direction – hence, their length has the same sign – and the operation switches to the concatenation mode as can be observed by

$$\text{concat}(a.b.\bot, \bot.a.b.c.d) = \bot.c.d$$

and

$$\text{concat}(\bot.a.b, y.x.a.b.\bot) = y.x.\bot$$

Finally, the consumption of a data path is not defined if the next step given by the second parameters is not matched by the first parameter. For instance, $\text{concat}(\bot.a, b.\bot)$ is undefined since it is not possible to walk a path up by a b-field step if you are at a point reached by addressing an a-field.

Based on those data path operators the interpretation of the *narrow* and *expand* operators is defined by

$$I[\mathit{ref.narrow}]((l, p_1), p_2, t) = (l, \text{concat}(p_1, p_2))$$

and

$$I[\mathit{ref.expand}]((l, p_1), p_2, t) = (l, \text{concat}(p_1, \text{inv}(p_2)))$$

Both operators are updating the data-path component of the input reference value (l, p_1) by a value computed based on the data path provided as a second argument.

Deref and Assign Operators Finally the effects of an operation reading a memory location ($ref.deref$) and updating a location ($ref.assign$) have to be formalized. When reading a value addressed by a reference (l, p) the value stored at memory location l has to be loaded and decomposed according to the data path p to reach the requested value. This decomposition is conducted by the get : $\mathcal{V} \times \mathcal{P} \to \mathcal{V}$ operation defined by

$$\text{get}(\nu, p) = \begin{cases} \nu & \text{if } p = \bot \\ \text{get}(\nu_i, \text{tail}(p)) & \\ & \text{if } \nu = [\nu_0, \ldots, \nu_n] \text{ and } \text{head}(p) = i \in \{0, \ldots, n\} \\ \text{get}(\nu_i, \text{tail}(p)) & \\ & \text{if } \nu = s[\ldots, n_i : \nu_i, \ldots] \text{ and } \text{head}(p) = n_i \in \mathbb{I} \\ \text{get}(\text{unpack}_t(\nu'), \text{tail}(p)) & \\ & \text{if } \nu = u[\nu'] \text{ and } \text{head}(p) = t \in \mathbb{T} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Based on the get operation the interpretation of $ref.deref$ can be defined by

$$I[ref.deref](e, (l, p)) = (e, \text{get}(\text{state}_e(l), p))$$

where $\text{state}_e(l) \in \mathcal{V}$ is obtaining the value stored at memory location l from the environment e and the application of get extracts the value of the sub-structure addressed by the reference (l, p).

Similar, when updating a the value stored within a memory location during an assignment operation, the current value has to be obtained, the addressed sub-structure located, and replaced by the new data element. The corresponding update-operation is based on the operator set : $\mathcal{V} \times \mathcal{P} \times \mathcal{V} \to \mathcal{V}$ defined by

$$\text{set}(\nu, p, v) = \begin{cases} v & \text{if } p = \bot \\ [\ldots \nu_{i-1}, \text{set}(\nu_i, \text{tail}(p), v), \nu_{i+1}, \ldots] & \\ & \text{if } \nu = [\nu_0, \ldots, \nu_n] \wedge \text{head}(p) = i \in \{0, \ldots, n\} \\ s[\ldots, n_i : \text{set}(\nu_i, \text{tail}(p), v), \ldots] & \\ & \text{if } \nu = s[\ldots, n_i : \nu_i, \ldots] \wedge \text{head}(p) = n_i \in \mathbb{I} \\ u[\text{pack}_t(\text{set}(\text{unpack}_t(\nu'), \text{tail}(p), v))] & \\ & \text{if } \nu = u[\nu'] \wedge \text{head}(p) = t \in \mathbb{T} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Finally the interpretation of the assignment operator is given by

$$I[ref.assign](e, (l, p), \nu) = (\text{set}(\text{state}_e(l), p, \nu), unit)$$

where $\text{state}_e(l)$ is obtaining the value of memory location l before the application of the operator, $\text{set}(\text{state}_e(l), p, \nu)$ is conducting the necessary update operation, $\nu \in \mathcal{V}$ is the value to be implanted into the addressed sub-structure of the referenced memory location's value and $unit$ is the single element of the unit domain.

Sources and Sinks An extension of the reference based modeling of mutable states covered within the previous section is based on the distinction of read-only, read-write and write-only memory locations. This concept is based on the three type families

$$\mathit{src}\langle t\rangle,\ \mathit{ref}\langle t\rangle,\ \text{and}\ \mathit{sink}\langle t\rangle$$

where $t \in \mathbb{T}$ is the type of the value stored in the referenced memory location. A source-reference of type $\mathit{src}\langle t\rangle$ is a read-only reference, a reference of type $\mathit{ref}\langle t\rangle$ is a read-write reference and a sink of type $\mathit{sink}\langle t\rangle$ can only be written. This is enforced by altering the types of the *deref* and *assign* operators to

$$\mathit{ref.deref} : (\mathit{src}\langle\alpha\rangle) \to \alpha$$

and

$$\mathit{ref.assign} : (\mathit{sink}\langle\alpha\rangle, \alpha) \to \mathit{unit}$$

Hence, only sources may be read and only sinks may be written. By adding the sub-type inference rules

$$\frac{}{\mathit{ref}\langle a\rangle <: \mathit{src}\langle a\rangle}\ \text{(src)}$$

and

$$\frac{}{\mathit{ref}\langle a\rangle <: \mathit{sink}\langle a\rangle}\ \text{(sink)}$$

turning references into special cases of sources and sinks, references can be read and written as usual. Additional modifications are required for the *ref.narrow* and *ref.expand* operators to support source and sink references as well, yet details are omitted for brevity at this point. The semantic interpretation of those operators, however, is not effected when treating sources and sinks equivalent to references – it is sufficient that the access privileges on the addressed memory locations are solely enforced by the type system.

Due to the requirement of interfacing with external C/C++ libraries depending on the distinction between const and non-const C pointers or C++ references, const type-modifiers have to be preserved when converting C/C++ input codes into INSPIRE and back. Therefor this extended variant of the mutable state extension is implemented within the Insieme Compiler.

Locks

A final common language extension required for the encoding of features encountered within a variety of parallel APIs are locks which might also be referred to as mutexs. A lock is represented utilizing the composed type

$$\mathit{lock} = \mathit{channel}\langle \mathit{unit}, 1\rangle$$

which is a channel with a capacity of 1 not relaying any information. Instead the synchronizing side-effects of *send* and *receive* operations are utilized to model *lock-acquire* and *lock-release* operations. The corresponding definitions are summarized by the following table:

Name	Type	Definition
lock.create	$() \rightarrow lock$	`() { auto res = channel.create(unit,1); channel.send(res, unit); return res; }`
lock.acquire	$(lock) \rightarrow unit$	`(lock l) { channel.recv(l); }`
lock.release	$(lock) \rightarrow unit$	`(lock l) { channel.send(l, unit); }`
lock.probe	$(lock) \rightarrow bool$	`(lock l) { return !channel.empty(l); }`

Upon creation the buffer of the underlying channel is filled with the unit element. The presence of this element marks the availability of the lock. To acquire the lock this unit-token needs to be retrieved (*channel.recv*), to release the lock some token needs to be returned (*channel.send*). The synchronizing side-effects are inherited from the utilized primitives.

3.9 Modeling Input Codes

To demonstrate the suitability of our language core and the associated extensions to model actual input codes a few example constructs are outlined within this sections. Although all of them have been implemented in the Insieme compiler the actual encoding is the topic of ongoing development and may have been further refined or even redefined to fit additional requirements. Also, the covered details shall only provide insights on how language constructs and API primitives are encoded into our IR and are by far not all the constructs supported by the Insieme project.

3.9.1 Sequential Host Language Constructs

The Insieme infrastructure is mainly designed to support parallel input codes based on C or C++ language extensions or APIs. Consequently our IR has to be suitable for modeling the constructs of those host languages as well. To provide a hint on how those might be encoded, the handling of some essential language features is briefly outlined in this section.

Variables

One of the most essential concepts of imperative languages, including C, are mutable variables. A basic variable declaration like

```
int x = 0;
```

is creating a mutable memory location (on the local stack) suitable to store a value of type **int**. The corresponding encoding in our IR is similar to

```
ref<int<4>> x = ref.var(0);
```

making the allocation of (stack) memory explicit. Also the type of variable x is $ref\,\langle int\,\langle 4\rangle\rangle$ indicates that the variable name x is actually referring to a memory location storing a value, not the value itself. Whenever the variable x is utilized within an operation, e.g. in the C code fragment

```
x + 1;
```

it is implicitly de-referenced to obtain its value for the computation. In our IR this is made explicit by

```
int.add(ref.deref(x),1);
```

which is equivalent to

```
*x + 1;
```

due to the syntactic sugar rules we have introduced earlier. Note that within C the type of the expression `x` and `x+1` is in both cases `int`. However, in the first case it is considered an *l-value*, hence suitable as a target for an assignment operation while in the second case it is a *r-value* – a value that can only be read. Within our IR, the expression `x` is of type `ref<int<4>>` identifying it as a suitable target for an assignment while `*x+1` is of type `int<4>`. Hence, by making the implicit creation and dereferenciation of memory locations inherent in C explicit within our IR, issues regarding l- and r-values have been bypassed. Also, implicit semantics introduced for the convenience of end users is made explicit, eliminating the requirement of analysis and other IR based utilities to consider those implicit effects and related distinctions.

Pointers

Beside basic (stack) variables data may be stored within heap allocated data structures. The language features provided by C to access data not necessarily located on the stack are C pointers. A code snippet like

```
int x = 0;
int* y = &x;
```

is declaring such a pointer (`y`) referencing the memory location of `x`. In our IR the same fragment could be represented by

```
ref<int<4>> x = ref.var(0);
ref<ref<int<4>> y = ref.var(x);
```

No new pointer construct is required. Instead a nested *ref* type is utilized for representing pointers. Also, the address-of operator `&` of C is not required since in IR the variable *x* is actually representing the memory reference, not the value as it is implicitly in C.

The double nesting corresponds to the fact that every C pointer is actually a mutable memory location (outer *ref*) containing the address of another location (inner *ref*). An operation updating the value of the referenced element like

```
*y = 12;
```

can therefore be encoded by

```
ref.assign(ref.deref(y), 12);
```

which is equivalent to

```
*y := 12;
```

by considering all the syntactic sugar defined above while an update of the actual pointer value like

```
int z = 14;
y = &z;
```

is encoded by

```
ref<int<4>> z = ref.var(14);
y := z;
```

While in the former case the location pointed to is updated in the latter the outer reference is modified.

The allocation of memory on the heap is realized in C by calls to library functions. An example is given by

```
int* z = malloc(sizeof(int));
```

which in our IR could be encoded using

```
ref<ref<int<4>>> z = ref.var(ref.new(int<4>));
```

Here the fact that in this declaration both, local stack memory and heap memory is allocated is made explicit. Also it shows that, unlike within C, no new construct had to be introduced to support the handling of data on the heap. C pointers are handled by utilizing the same reference type extensions utilized for modeling any other mutable state in the IR.

Pointer Arithmetic Besides addressing and manipulating data stored in arbitrary locations (heap, stack, or even others) C pointers provide an additional feature – *pointer arithmetic* and their connection to arrays, which are indeed pointers. For instance, the following code snippet outlines several common ways of dealing with arrays and pointers in C

```
1   int x[5];           // array of 5 ints on the stack
2   x[3];               // reads 4th element of x
3   *x;                 // reads 1st element of x
4   *(x + 3);           // reads 4th element of x
5
6   int* y = x;         // y is an alias for x
7   y[3];               // y can be used like an array
8   *(y + 3);           // or pointer
9
10  int* z = malloc(sizeof(int) * 5);  // on heap
11  z[0] = 12;          // used like an array
12  *(z + 3) = 14;      // or pointer (4th element)
13
14  int* w = z + 2;     // an offseted alias to array z
15  *w;                 // reads 3rd element of z
16  w[1];               // reads 4th element of z
17
18  w--;                // offset is moved down by one
19  *w;                 // reads 2nd element of z
20  w[1];               // reads 3rd element of z
```

In the first block an array x is created on the stack and accessed utilizing array (line 2) and pointer notation (line 3 and 4). In the following block an alias y for x is created (line 6) which can be accessed utilizing the same notation (line 7 and 8). This illustrates the duality of arrays and pointers in C as well as the mechanism utilized when passing arrays as arguments to function calls. However, note that although the same operator is utilized in line 2 and 7 the execution of line 2 involves a single memory access while line 7 requires two operations – reading the value of variable y to compute the address of the location containing the actual value to be read.

In the following block a heap allocated array is created (line 10) and accessed again utilizing array (11) and pointer notation (12). In both cases 2 memory access operations are involved. Finally, within the last block, a pointer w representing an offseted alias of array z is created, read and manipulated – another common feature frequently utilized within C codes.

Following the naive encoding of pointers covered above, pointer arithmetic can not be supported based on the available primitives. To support pointer arithmetic, pointers needed to be encoded as pairs of the type

```
struct { ref<array<α>> base; int<4> offset; }
```

where α is to be substituted by the element type, base is a reference to an allocated memory location and offset an implicit offset to be incorporated whenever the pointer is dereferenced. For brevity, in the following code snippets we will utilize the abbreviation *ptr* for this generic struct-based pointer type. Further we introduce the derived operators

Name	Definition

ptr.init	`(ref<array<α>> r) → ptr {`
	  `return struct { base = r, offset = 0 };`
	`}`

ptr.read	`(ptr p) → ref<α> {`
	  `return array.ref.elem(p.base, p.offset);`
	`}`

ptr.add	`(ptr p, int<4> i) → ptr {`
	  `return struct {`
	    `base   = p.base,`
	    `offset = p.offset + i`
	  `};`
	`}`

to initialize, read and update pointers.

Based on those the first block of the C code example above can be encoded in our IR by

```
ref<array<int<4>>> x
        = ref.var(array.create(int<4>,5));
ref.deref(array.ref.elem(x),3);
ref.deref(ptr.read(ptr.init(x)));
ref.deref(ptr.read(ptr.add(ptr.init(x),3)));
```

In this encoding the array x is allocated on the stack – due to the $ref.var$ call – and treated like a standard array. However, when switching to a pointer notation the reference to the array is converted into the pair-based pointer encoding utilizing a call to *ptr.init*. Based on this representation pointer arithmetic is performed. Finally, whenever read, the pointer encoding is unwrapped by a call to *ptr.read* and the addressed memory location is dereferenced using the standard $ref.deref$ operator.

The second block creating the alias y of x

```
int* y = x;        // y is an alias for x
y[3];              // can be used like an array
*(y + 3);          // or pointer
```

is encoded into INSPIRE similar to

```
let ptr_t = struct {
  ref<array<int<4>>> base;
  int<4> offset;
};

ref<ptr_t> y = ref.var(ptr.init(x));
ref.deref(ptr.read(ptr.add(ref.deref(y),3)));
ref.deref(ptr.read(ptr.add(ref.deref(y),3)));
```

As can be observed, the array subscript based access (*y[3]*) including an implicit dereferencing and the pointer based access (*∗(x+3)*) are represented by the same IR expression. Also the two involved memory read operations are explicitly represented by the two invocations of the *ref.deref* operator while in the previous direct array access case only one read operation was involved. The same is true for the heap based operations of the third block, where

```
int* z = malloc(sizeof(int) * 5);  // on heap
z[0] = 12;          // used like an array
*(z + 3) = 14;      // or pointer (4th element)
```

is encoded into

```
let ptr_t = struct {
  ref<array<int<4>>> base;
  int<4> offset;
};

ref<ptr_t> z = ref.var(ptr.init(
    ref.new(array.create(int<4>,5))
));
ptr.read(ptr.add(ref.deref(z),0)) := 12;
ptr.read(ptr.add(ref.deref(z),3)) := 14;
```

where a new array is allocated on the heap (call to *ref.new*) and its location is stored in a local mutable variable *z*. As in the previous case, the array and pointer based access to the array elements is encoded utilizing equally structured expressions.

Finally, the remaining pointer arithmetic operations covered by the last two blocks of the C example above given by

```
int* w = z + 2;   // an offseted alias to array z
*w;               // reads 3rd element of z
w[1];             // reads 4th element of z

w--;              // offset is moved down by one
*w;               // reads 2nd element of z
w[1];             // reads 3rd element of z
```

including the creation of a pointer referencing an array with a given offset, is represented within our IR by

```
ref<ptr_t> w = ref.var(ptr.add(ref.deref(z),2));
ref.deref(ptr.read(ref.deref(w)));
ref.deref(ptr.read(ptr.add(ref.deref(w),1)));

w := ptr.add(ref.deref(w),-1);
ref.deref(ptr.read(ref.deref(w)));
ref.deref(ptr.read(ptr.add(ref.deref(w),1)));
```

making all the implicit and explicit operations involved in the C code explicit utilizing a unified set of primitives.

Additional pointer arithmetic operations, including comparison operators and pointer differences, can be encoded using similar means by focusing on the offsets. This encoding is exploiting the fact that these kind of operators have undefined behavior when being applied to unrelated pointers according to the C language specification.

However, although encoded as pairs within the IR, in the backend these pairs of base pointers and offset values and their associated operators are converted back into regular C pointer constructs. The purpose of the encoding is mere to simplify and unify the handling of memory references and locations in the IR and its associated analysis and transformation utilities.

Global Variables

Unlike within processed input codes our IR does not exhibit a concept similar to a global scope since every representation of a full program is only consisting of a single expression. However, although their usage is discouraged, global variables are a frequently encountered construct utilized by many programs. Hence, support for those is required.

Fortunately global variables can be easily encoded in our IR utilizing literals. For instance, a global counter c as utilized in

```
int c;
void init() { c = 0; }
int inc() { return ++c; }

void main() {
  init();
  inc();
}
```

can be encoded within INSPIRE using a simple literal

```
  lit(c:ref<int<4>>)
```

utilized by the corresponding implementations. Essentially the literal represents a named global memory location containing an integer. The full encoding of the given example looks like

```
()→unit {              // encoding of main
  ()→unit {            // encoding of init
    c := 0;
  }();                 // init call
  ()→unit {            // encoding of inc
    c := c+1;
    return c;
  }();                 // inc call
}
```

where the literal c is defined as introduced above.

3.9.2 Common Parallel Constructs

A variety of parallel APIs offer similar primitives and building blocks which we will describe at this point before actually diving into concrete APIs in order to reduce the amount of details to be covered in later sections.

Barriers

Barriers are among the basic primitives available in many APIs, in particular including MPI, OpenMP and OpenCL. Barriers allow groups of threads to synchronize. The group management is handled within INSPIRE by the creation of thread groups utilized for processing jobs. To realize a barrier operation among the members of a group an operator *barrier* is defined by

```
()→unit {
  redistribute(unit, (α d)→unit { return unit; });
}
```

which simply utilizes the blocking data-sharing construct ***redistribute*** for realizing a synchronization event between threads of a group without actually exchanging any data. The data item contributed by each thread is the *unit* constant and the selection function is ignoring the aggregated array of contributions and just returning the *unit* value. Consequently the only effect of an invocation of the *barrier* operator is its synchronization effect as desired.

Reductions

Another frequently encountered class of primitives are parallel reduction operations. In such an operation every thread contributes a value which will then be aggregated utilizing some, typically associative and commutative, binary operator. For the encoding of such reduction operations a generic function *reduce* of type

$$(\alpha, (\beta, \alpha) \to \beta, \beta) \to \beta$$

has been introduced where the first parameter is the value contributed by the current thread, the second the reduction operator and the third the initial value for the reduction, e.g. the identity element of the reduction operator. It is implemented by

```
(α v, (β,α)→β op, β init)→β {
  redistribute(v,(array<α> data)⇒{
    ref<β> res = var(init);
    for(uint<8> i = 0 .. getNumThreads(0)) {
      res := op(*res, *(data[i]));
    }
    return *res;
  });
}
```

which is simply applying the reduction operation on the aggregated list of contributions collected by the redistribute operator. An example application summing up all the values of thread-local variables x among the threads of a thread group looks like

```
reduce(x, int.add, 0);
```

where x is the variable to be contributed by the local thread, *int.add* the operator to be utilized for the reduction and 0 the literal encoding the identity element of the reduction operation. The result will be the sum of all the contributions of the individual threads and as a side effect the synchronization of all threads.

3.9.3 Parallel APIs

Finally we can provide an overview on the actual encoding of parallel constructs encountered in our primary targeted parallel APIs.

OpenMP

The central element within OpenMP to express parallelism is the definition of a parallel region similar to

```
#pragma omp parallel
{
  ...
}
```

which is represented based on our IR's primitives by

```
merge(parallel(job[1,INT_MAX]()⇒{
  ...
}));
```

Hence, a parallel region is outlined into an extra function by wrapping it into a closure utilizing a *bind expression*, wrapped into a job to be processed by at least one thread and spawned by a call to the ***parallel*** primitive. The resulting thread group is merged since the spawning thread in OpenMP is blocked until the nested parallel region has completed its task.

In case OpenMP *data clauses* are specified corresponding code has to be introduced. For instance, the input code

```
int x = 0;
#pragma omp parallel firstprivate(x)
{
  x = x+1;
}
```

is converted into

```
ref<int<4>> x = var(1);
merge(parallel(job[1,INT_MAX]()⇒{
  ref<int<4>> px = var(x);
  px := *px + 1;
}));
```

which is creating a private copy *px* of the otherwise shared variable *x* in the body of the parallel job. This private copy is then utilized for the applied operations as defined by the OpenMP standard. Similar approaches are followed to provide support for *private* and *lastprivate*.

Worksharing Constructs OpenMP offers three *worksharing constructs* for C – the *loop construct*, *sections* and the *single* construct. All of them are encoded within our IR using its unified worksharing construct ***pfor***. For starters, a parallel OpenMP loop like

```
#pragma omp for
for(int i = 0 ; i != 10; i++) {
  <loop body>
}
```

is represented by

```
pfor(0,10,1,(int<4> a, int<4> b, int<4> c)⇒{
  for(int<4> i = a .. b : c) {
    <loop body>
  }
});
barrier();
```

The encoding is based on an outlined version of the original loop body where all required information is captured by the utilized bind expression. Within this function the original loop body is processed. Since our IR's ***pfor*** operator does not cause an implicit synchronization an explicit *barrier* call is added at the end – unless an explicit OpenMP *nowait* tag is provided in the input code.

In case a for loop is conducting a reduction the corresponding IR operator introduced in the previous section is utilized. For instance, the simple case

```
int sum = 20;
#pragma omp for reduce(+:sum)
for(int i = 0 ; i != 10; i++) {
  sum += a[i];
}
```

is internally represented by

```
ref<int<4>> sum = var(20);
{
  ref<int<4>> psum = var(0);
  pfor(0,10,1,(int<4> a, int<4> b, int<4> c)⇒{
        for(int<4> i = a .. b : c) {
            psum := *psum + *a[i];
        }
    });
    sum := *sum + reduce(*psum,int.add,0);
}
```

including the creation of a private copy *psum* of the variable *sum* initialized with the identity element of the reduction operator + (=0), the internal, private aggregation while processing loop iterations and the concluding group-wide aggregation utilizing the reduce operator. Note that the application of this final operator is causing an implicit synchronization among the threads of the group and hence allows the obligatory *barrier* to be omitted.

OpenMP sections are also distributed among the threads of a group by converting them into iterations of a parallel loop and selecting sections based on the value of the iterator variable. Similarly the *single* construct is interpreted like a single section.

Synchronization OpenMP barriers are realized utilizing the *barrier* function introduced in the previous section. However, OpenMP also provides additional synchronization means, in particular *critical regions*. Those are implemented by utilizing global variables and the *lock* extension which itself is based on IR channel operations.

For example, two critical regions

```
#pragma omp critical
{ <A> }
...
#pragma omp critical
{ <B> }
```

are encoded by

```
lock.acquire(g_omp_critical_lock);
{ <A> }
lock.release(g_omp_critical_lock);
...
lock.acquire(g_omp_critical_lock);
{ <B> }
lock.release(g_omp_critical_lock);
```

where *g_omp_critical_lock* is a literal of type *lock* which is equivalent to a global variable in a conventional context. In case the critical section is named a fresh global lock is bound to the name and utilized accordingly.

Tasks Another frequently utilized feature in OpenMP are its tasks. Their encoding is similar to the encoding of tasks in Cilk which will be covered next.

Cilk

Essentially Cilk adds two additional keywords to its C host language: *spawn* and *sync*. Both have to be encoded into IR constructs. The example code fragment

```
x = spawn f(n-1);
y = spawn f(n-2);
sync;
```

is internally represented by

```
parallel(job[1,1]()⇒{
  x := f(n-1);
});
parallel(job[1,1]()⇒{
  y := f(n-2);
});
merge();
```

which is creating jobs to be processed by a single thread for each spawned procedure call. As required by the Cilk specification, an execution in an other thread is not mandatory. The presents of the ***spawn*** keyword is merely indicating the potential of concurrently processable workload. The concluding synchronization is conducted by an invocation of the ***merge*** operator which is equivalent to Cilk's ***sync*** operator. Furthermore the implicit ***sync*** operation at the end of every task-spawning function in Cilk is made explicit. The OpenMP *task* and *taskwait* constructs are encoded equivalently.

OpenCL

When running an OpenCL kernel, a function is being applied to every node within a 1, 2 or 3-dimensional grid in parallel. Thereby, the total grid, known as the *global range*, is partitioned into a large number of equally sized *local groups* of *work items*. Items within the same local group may share data using a fast shared memory segment. Also, synchronization means are exclusively offered within local groups.

This two-level decomposition and its various aspects need to be reflected within the IR since its proper utilization is the key for high-performance computing on accelerators. The two levels are encoded using two nested thread groups. The following code fragment illustrates the general schema:

```
<global-vars>
merge(parallel(job[<global-range>]()⇒{
  <local-vars>
  merge(parallel(job[<local-range>]()⇒{
    <kernel-code>
  }));
}));
```

The outer level corresponds to the global range – reduced to a single dimension if applied to a 2 or 3-dimensional grid – using one thread per local group and the inner level models the local groups. Global and local memory is allocated for the corresponding variables within their code sections respectively. Within the actual kernel code, $getThreadID(0)$ can be used to obtain the index of the work item within the local group and $getThreadID(1)$ to access the group ID.

The real challenge when converting OpenCL codes into IR is the rather extensive "boilerplate" code for device setup and kernel preparation surrounding an OpenCL kernel invocation. In the Insieme compiler the host code is analyzed to identify the kernel calls and the associated data manipulation operations to generate IR code focused on those essential operations. The original OpenCL library calls are thereby eliminated by the frontend for clarity and re-added by the backend using runtime system APIs.

The seamless integration of the kernel code into the host program enables the compiler to analyze and tweak the execution of the entire program (host and kernel part). For instance, constants may be forwarded from the host to the kernel codes or successive kernel invocations operating on the same data may be merged. Also, the data a kernel is applied to may be automatically split and distributed among multiple devices [36].

MPI

Unlike for OpenMP, Cilk and OpenCL, in an MPI application the parallelism is neither confined to a single code region nor a single process. Fortunately, our parallel model is not restricted to threads within a single process. IR threads of a group can be distributed among multiple processes, as long as they do not share access to a common memory location – as it is always the case for MPI codes.

The whole-program parallelism of an MPI application can be modeled using a top-level ***merge*/*parallel*/*job*** combination. Within the executed job, an array of communication channels of type $channel\,\langle msg, 1\rangle$ is created (COM-array). The utilized msg type is a abstract type modeling MPI messages and associated meta data, including their type, size and actual payload. The basic structure of an MPI program in INSPIRE is the following

```
merge(parallel(job[1,INT_MAX]()⇒{
   array<channel<msg,1>> com =
      redistribute(channel.create(msg,1), id);
   /* ... MPI program body ... */
   channel.release(com[getThreadID(0)]);
}));
```

where id is the identity function of type $(\alpha) \rightarrow \alpha$. This frame covers the obligatory MPI_Init and $MPI\_Finalize$ calls. However, unlike those it can be nested and processed multiple times.

Each channel within the COM-array is associated with one of the participating workers. If one MPI worker sends data to a peer using `MPI_Send` or similar primitives, the data is encapsulated into a message and submitted to the channel associated with the receiver using the *channel.send* primitive. Consequently, MPI instructions receiving messages (e.g. `MPI_Recv`) are based on the *channel.recv* operator accessing the channel associated to the local thread. All point-to-point MPI communication routines, including the asynchronous variants, can be modeled similarly.

Beside Send/Recv routines, MPI covers a rich set of collective operations. These kind of operations have been generalized by the IR's *redistribute* operator which is specialized for the particular cases. Examples covering barriers and reduction operators have been demonstrated above. Another example would be an MPI broadcast operation distributing information from process 0 to all other involved processes. Such an operation is modeled by

```
// setup
ref<int<4>> x = var(0);
if(getThreadID(0) == 0) {
  x = ... ;    // some source at process 0
}

//  broadcast (0-to-all)
x := redistribute(*x,(array<int<4>> d)→int<4> {
  return d[0];
});
```

Similar derived constructs can be used to represent other *broadcast*, *scatter*, *gather* and *all-to-all* routines. However, for supporting more advanced features offered by MPI, including user defined data types or the utilization of message tags or message source IDs the necessary details in the encoding are clearly beyond the scope of this summary as well as the subject of ongoing research.

Remark: Converting shared to distributed memory code is known to be a hard problem. Presenting instances of the problem in a different format can obviously not eliminate the underlying fundamental complexity. With our work we do not claim to provide any solutions for this problem. The goal of our IR is to offer one common infrastructure enabling the representation of both kind of applications to facilitate future research in this and other directions.

Other Languages

The parallel control-flow model of INSPIRE is generic enough to support a variety of additional parallel APIs, including *pthreads* and the C++11 threading facilities. Also, PGAS based approaches should be coverable by our IR. However, this issue has not yet been sufficiently investigated since no language based on this paradigm has been integrated into our system so

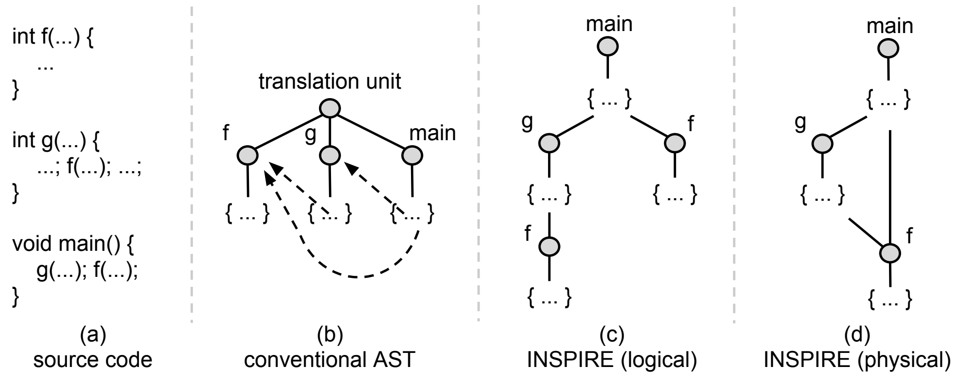


Figure 3.6: Comparison of IR structures.

far. Furthermore, our IR is not limited to C/C++. In particular our platform can serve as the foundation for implementing parallel domain specific languages (DSLs) which would benefit from the available infrastructure and from the implicit mapping of parallel constructs to sophisticated runtime system primitives within our backend.

3.10 Implementation

So far our focus has been solely placed on language aspects of INSPIRE. No implementation details regarding the actual data structures utilizes for building the internal program representation within the Insieme compiler have yet been covered. While the full details are clearly beyond the scope of this thesis – and are best studied by investigating the actually implementation – this section will focus on a few key implementation details essential for the following chapters. In particular the organization of the IR data structure itself and its implications on the way sub-structures can be addressed and manipulated shall be covered.

3.10.1 Overall Structure

As has been mentioned earlier, unlike other high-level IRs, INSPIRE does not strive to reflect the structure of translation units. Figure 3.6 illustrates how the C source depicted in (a) is represented. Within a typical AST based IR, a data structure representing a translation unit is created (b). The root node represents an organizational structure, e.g. a file, and lists a set of top-level definitions. Each definition is defined by sub-structures which may refer to top level elements or even external content.

INSPIRE follows a different approach. Instead of reflecting the organization of the input source files, it models the actual execution using a single expression. Figure 3.6 (c) illustrates the corresponding parse tree. The root element represents the entire execution of a code fragment, hence, typically the main function of a standalone application or some isolated library routine. Furthermore, whenever a function is called, the expression defining the target function is present right at the call site, independently from the translation unit it was originally defined in. This way, our IR provides a holistic view on the entire execution of a program, facilitating context sensitive or even whole-program optimizations.

Since all IR codes are simple terms of the grammar introduced in Section 3.4, the data structure utilized for representing instances is a plain tree structure covering expression, statement and type constructs equally, not containing any cross or back edges and reflecting the recursive definition of all our language constructs. The only addition to the grammar of Definition 3.1, 3.4 and 3.6 is the decision to annotate every expression explicitly with its *most general type* (see Definition 3.29) since it is a frequently utilized property of expressions.

The self-contained design of our IR allows for a variety of properties to be deduced from a given sub-tree without the requirement of any additional global context. Also, local modifications to a function f, valid only within the current call-context (reflected by the path from the root node to the function), do not affect any other instance of f.

The downside of a representation following the schema of Figure 3.6 (c) is the huge memory requirement due to the excessive duplication of functions and types. To alleviate this problem, nodes are shared in our implementation as illustrated in Figure 3.6 (d). The logical tree structure is physically realized by a DAG. This enables the representation of IR codes consisting of several million logically addressable nodes (by their path from the root) using a few thousand physical nodes. For example, BT, the largest NAS benchmark [9] in terms of lines of code (2.281), is encoded in our IR using 10.296.189 addressable logical nodes, yet physically realized using only 12.549 nodes, resulting in a total memory consumption of 1.8 MB.

Annotations

The information stored within the IR DAG is limited to the language constructs covered by the grammar definition of section 3.4. However, in some cases, additional information like user defined hints on loop scheduling policies, results of analyzes, or assertions on values may have to be stored somewhere. To provide an integrated means to record this kind of information, every IR node can be annotated with generic information. Beside being utilized internally for caching information, these annotations are also utilized for forwarding meta-information on constructs through various stages of the compiler and even to the runtime system.

If utilized properly, annotated information representing results of analysis never has to be updated due to modifications of the structure it has

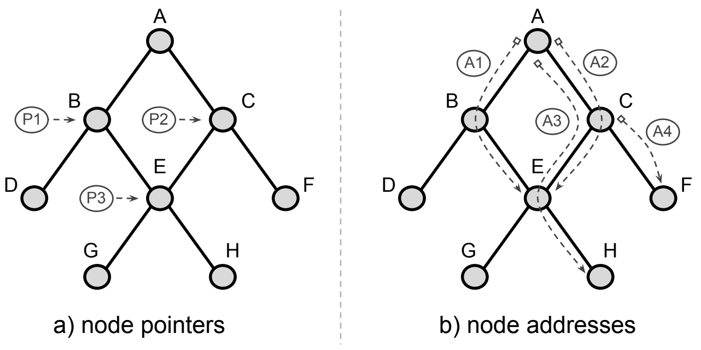


Figure 3.7: Two ways of addressing substructures within an IR DAG.

been attached to since all IR nodes are immutable. Furthermore, due to the implicit node sharing, analysis results for identical substructures are implicitly available within all contexts the corresponding IR substructure is utilized. Annotations therefore provide a convenient infrastructure for implementing memorization for tools operating on IR structures which can provide tremendous performance benefits.

3.10.2 Addressing Substructures

There are two essential ways of addressing IR structures. Both are illustrated in Figure 3.7. The first, simple approach is by utilizing a simple *pointer* to the root of an IR substructure (DAG) to address the corresponding fragment. Consequently we refer to this concept as *node pointers*. For instance, the pointer P1 in Figure 3.7 a) references the sub structure

$$B(D, E(G, H))$$

while P2 addresses

$$C(E(G, H), F)$$

While simple pointers are a very common and straight-forward means in a C++ data structure to address sub-structures they have their limitations. For instance, if we consider the full structure which is encoding the term

$$A(B(D, E(G, H)), C(E(G, H), F))$$

pointers can not be utilized to address the first appearance of the sub-term

$$E(G, H)$$

since this sub-term is shared. Pointer P3 in Figure 3.7 a) is addressing the proper sub-term but the multiple appearances of the same sub-term can not be distinguished due to the implicit sharing of nodes. However, for various utilities, in particular tools to transform IR structures, such a distinction is necessary. Also, from a given pointer one can not navigate to its *parent node* since within the physical IR DAG there is no unique parent in the general case. A parent would only exist in the logical IR tree, which is not materialized. Consequently a second way for addressing substructures satisfying these requirements had to be introduced.

So called *node addresses* do not only reference individual nodes within the IR structure, they describe the path to those nodes starting from an arbitrary *root node*. Figure 3.7 b) illustrates this approach. Addresses A1 and A2 are both referencing the sub-term

$$E(G, H)$$

of the overall structure. However, while A1 is referencing the first appearance within

$$A(B(D, E(G, H)), C(E(G, H), F))$$

reached over the substructure $B(D, E(G, H))$ the address A2 references the second by describing the path over $C(E(G, H), F)$. Correspondingly address A3 is referencing the H in the second appearance of $E(G, H)$.

Addresses are not required to start at a fixed, common root node. Like file system directory paths they may start at an arbitrary node and describe a relative path to one of its sub-nodes. Address A4 does so by referencing the F in $C(E(G, H), F)$ where the latter is the root node of the address.

Implementation wise an address is realized by a consistent sequence of *node pointers*. However, to minimize the overhead of handling *node addresses* common prefixes of addresses are shared. Hence, an address $[a, b, c, d]$ references the address $[a, b, c]$ as a base address and extends it by an extra step to the node referenced by d.

3.10.3 Manipulating Substructures

A obvious problem of immutable data structures is that they can not be altered when conducting transformations. For instance, a simple term

$$A(B(D), C(D))$$

can not be transformed into

$$A(B(D), C(E))$$

by addressing the second D and simply rewriting it to an E. Instead new structures forming a proper representation of the desired, modified term need to be constructed in a bottom up fashion. Thereby common sub-structures present in the original term are supposed to be reused. The corresponding operation is illustrated in Figure 3.8. The *in* pointer is referencing the

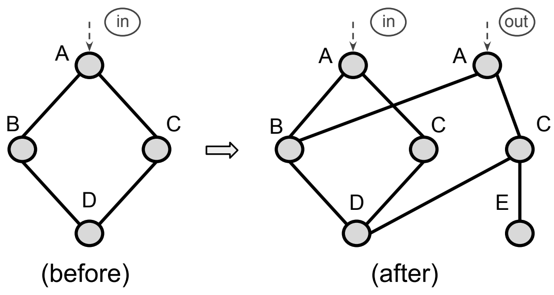


Figure 3.8: Applying transformations on an immutable IR DAG.

original structure and *out* the transformed version. As can be observed, common sub-terms are shared between both instances. Furthermore, based on this infrastructure no manipulation operation is destructive. The original code versions are preserved without any extra effort – in particular not the requirement of creating a deep copy of the structure.

However, due to this design decision, whenever a node is going to be altered new versions of all nodes along the path between the updated node and the root node representing the handled code fragment have to be created. Yet, the length of this path is logarithmic in the size of the overall program and therefore imposes an acceptable small price for a easy-to-use, memory efficient data structure with implicit sharing of annotated information.

Developers utilizing the IR data structures are shielded from those internal mechanisms by higher level manipulation utilities abstracting this low level IR structure organization. The implicit sharing of nodes and the creation of modified version of existing nodes when transforming IR structures is hidden from the developer.

3.11 C++ Support

So far the content covered within this chapter focused on supporting C constructs for brevity. However, within the Insieme project, INSPIRE has been adapted to support C++ codes as well. An overview on the necessary additions shall be provided in this final section to complete the coverage of the Insieme IR. While most of the content presented in this chapter has been implemented within the Insieme infrastructure as described, some of the details may still be the subject of ongoing or future developments.

3.11.1 Challenges and Requirements

In conventional compilers support for C++ is achieved by implementing frontents lowering C++ input code into classic, byte-level intermediate representations. Structural information regarding e.g. classes, constructors, destructors, (virtual) member functions, inheritance relations, templates, temporary variables and C++ references are thereby resolved. Hence, non of those concepts are present within the utilized low-level IRs. On the one hand this greatly simplifies the handling of program code due to significantly reduced complexity. On the other hand, similar to the problem of decomposing loops into jump instructions, the structural information regarding type relations or virtual member function calls is lost, effectively hampering the ability of a compiler to analyze and tune these kind of language features.

Also, for low-level IRs, the scopes of "global" operations and transformations, e.g. well known techniques like *global value numbering* utilized for *common sub-expression eliminations*, are individual procedures. However, in C++ programs, built based on an *object oriented* programming style, computations are typically distributed among a plethora of member functions invoking each other through dynamic dispatching mechanisms. Interprocedural analyzes and the consideration of the possibility of dynamically dispatched function calls gain a much higher relevance when dealing with C++ applications than they do within procedural C codes. In the latter, computational intensive code is typically contained within a single function and the problem of dynamically dispatched functions simply does not exist since it is not supported by the C language.

For these and other reasons the Insieme compiler is utilizing a high-level IR preserving a variety of constructs of the input program. However, according to our design principles, the incorporation of C++ support should be realized by adding as little extra constructs to our intermediate language as necessary. One common approach that has been followed in the past is to rewrite C++ applications into C applications before converting them into the compiler internal IR. Although this would essentially reduce the number of constructs to be added to the IR to zero, this approach induces similar issues than a lowering to low-level IRs, yet on a higher level. For instance, the naive encoding of high-level features, in particular virtual function dispatching or support for *Runtime Type Information* (RTTI) would result in complex C code including fragments that are hardly processable by compiler analysis. A careful encoding considering the compilers analytical abilities is required to mitigate those effects [114].

Still, the lowering of C++ code to C code imposes another, even severe problem. One of our design objectives is to provide support for an open system. In particular the Insieme compiler is required to support interfacing external libraries not processed by our compiler. Hence it has to be possible to model interfaces of those external modules such that they can be addressed and utilized by IR internal code fragments. Pure C code can not address member functions of objects described by external libraries – unless the *name mangling* is reproduced in C – or utilize templated code as encountered within most modern C++ libraries including the ubiquitous

Standard Template Library (STL) being part of the C++ program language specification and frequently referenced third-party libraries including *boost*.

In particular standardized template libraries implementing widely utilized container classes including vectors, sets and maps may even be preferably treated as abstract data types within a high-level compiler – just as end users are used to utilize them. For instance the semantic of the operations defined on a C++ *vector* is well defined and may be incorporated into an analysis instead of trying to deduce the same information from its implementation. This additional awareness of higher-level constructs can lead to more accurate static program analyzes. Consequently the requirement on the IR of the Insieme compiler is to provide means to interface with abstract external C++ (template) libraries such that end user code based on those can be processed independently.

Requirements

C++ comprises a plethora of language features providing end users the means for building flexible, reusable code. Fortunately only a subset of those features are required for interfacing with C++ codes. These include:

- **Constructors, Destructors and Member Functions** – although essentially just ordinary functions with an extra implicit *this* parameters those functions require a special syntax when being invoked and therefore need to be marked as such explicitly. Also, the implicit invocation of destructors at the end of local scopes has to be incorporated.

- **Subtype Relations** – in addition of providing means to C++ to inherit implementations, C++ class inheritance also introduces subtype relations between types. In the general case a class *B* derived from a class *A* is a sub-type of *A* and may be utilized as a substitute in a variety of contexts. Hence, the inheritance relation needs to be covered in the IR as well.

- **Virtual Member Functions** – another aspect of the polymorphism encountered in object-oriented programming languages which allows member functions to be overloaded based on the runtime type of the object they are applied to. Means for a corresponding mechanism are essential for supporting object-oriented C++ applications to be processed by the Insieme compiler.

- **Templates** – among the most powerful features of C++ for implementing flexible, reusable libraries are its template facilities. It supports the definition of code templates to be instantiated for specific types. Most standard and third-party libraries are utilizing this mechanism – sometimes exclusively – for providing implementations of data structures and algorithms. Support for corresponding interfaces is hence necessary.

- **Concepts** – an implicit trait of C++ is that classes utilized as template parameters are required to support a certain set of named operators. For instance, *std :: set* demands, among others, an implementation of the < (less-than) operator of potential element types. Means for attaching such *named* operators to types are required.

- **C++ References and other Type Constructs** – since C++ references are essentially pointers which are implicitly dereferenced whenever utilized they could be modeled within the IR as such, making dereferencing operations explicit. However, the distinction between pointers and C++ references may influence the resolution of overloaded functions when being passed as an argument. So do *const* modifiers and *enums* – which hence have to be preserved when encoded into our IR.

- **Exceptions** – on the statement level C++ adds support for exceptions being raised and caught as part of the control flow. As for the other features, since external libraries may utilize exceptions to communicate with the client code, corresponding mechanisms are required to be supported by INSPIRE.

Although this list seems comprehensive it still eliminates a variety of C++ constructs that would be encountered within a conventional, complete C++ AST like the clang AST [3]. Those include access modifiers, (most) names, namespaces, name resolution, function overloading, default parameters, implicit conversions, initializer lists, elidable constructor calls, creation and destruction of temporary objects, the distinction between declarations and definitions, template specializations, template meta programming, and the handling of translation unit boundaries. Within the Insieme compiler all these are handled by the frontend and/or converted into equivalent code utilizing the available set of IR constructs.

3.11.2 Language Modifications

To provide proper support for C++ language features a few additional type, expression and statement constructs have been added to the language covered within section 3.4.

Types

To add support for objects, some modifications and extensions to the type constructs listed in Definition 3.1 needed to be conducted. Those modifications add support for modeling classes to the IR type system and introduce additional function types to distinguish *constructors*, *destructors* and *member functions.*

Classes The representation of classes is realized by extending the *struct* type constructor by the capability of referencing base types to be extended. The rule (struct) of Definition 3.1 is therefore extended to the following format:

Definition 3.55 (classes)**.** Let $p_1, \dots, p_n \in \mathbb{T}$ be distinct types, $t_1, \dots, t_m \in \mathbb{T}$ be types, $n_1, \dots, n_m \in \mathbb{I}$ be identifiers utilized as names and $v_1, \dots, v_n \in \mathbb{B}$ be boolean flags for any $n, m \in \mathbb{N}$. Then

$$struct \ : \ v_1 \ p_1, \dots, v_n \ p_n \ \{n_1 : t_1, \dots, n_m : t_m\}$$

is a struct type inheriting fields from the types $p_1, \dots, p_n$ and defining the additional fields $n_1, \dots, n_m$ of type $t_1, \dots, t_m$.

The flags $v_1, \dots, v_n$ determine whether the inheritance is *virtual*, hence multiple references to the same base class within the inheritance DAG are shared, or not, causing an individual copy of each base class to be present. A struct type r is a sub-type of any type $b \in \mathbb{T}$ whenever b only occurs once within r's closure of the inheritance relation.

Abstract Classes To model external libraries, abstract types of the form (abstract) of Definition 3.1 are utilized. Although providing the generic, abstract properties required to model external (templated) classes encountered within C++ libraries, abstract types as covered within Definition 3.1 lack the ability of describing inheritance relationships. To add this feature the rule (abstract) of Definition 3.1 is extended similar to the struct type rule as follows:

Definition 3.56 (abstract class type)**.** Let $i \in \mathbb{I}$ be an identifier, $p_1, \dots, p_n \in \mathbb{T}$ be distinct types, $v_1, \dots, v_n \in \mathbb{B}$ be boolean flags and $t_1, \dots, t_k \in \mathbb{T}$ be types for any $n, k \in \mathbb{N}$. Then

$$i \ : \ v_1 \ p_1, \dots, v_n \ p_n \ \langle t_1, \dots, t_k \rangle$$

is an abstract type extending the base types $p_1, \dots, p_n$.

The flags $v_1, \dots, v_n$ have the same effect as for the struct type, namely to distinguish virtual and non-virtual inheritance. As for structs, abstract types are implicit sub-types of any type occurring once within the closure of the inheritance relation seeded by themselves.

Definition 3.57 (class types)**.** Let $\mathbb{C}$ be the union of all struct or abstract types, type variables or recursive types which are equivalent to struct types. We refer to this set of types as *class types.*

Constructor, Destructor and Member Function Types In addition to the extended versions of the struct and abstract type construct three new type constructs representing the type of constructors, destructors and member functions have been introduced.

Definition 3.58 (object function types)**.** Let $c \in \mathbb{C}$ be a class type and $t_1, \ldots, t_n, t_r \in \mathbb{T}$ be types. Then the constructs

$$c :: (t_1, \ldots, t_n) \rightarrow t_r \tag{mfun}$$
$$c :: (t_1, \ldots, t_n) \tag{ctor}$$
$$\sim c :: () \tag{dtor}$$

are types as well.

The class type c determines the type of object corresponding functions can be applied on. The types $t_1, \ldots, t_n$ define the parameters to be passed as arguments and the type t_r determines the result type of a member function invocation. For the type deduction types of the shape

$$\begin{aligned} &c :: (t_1, \ldots, t_n) \rightarrow t_r \\ &c :: (t_1, \ldots, t_n) \\ &\sim c :: () \end{aligned}$$

are considered equivalent to the function types

$$\begin{aligned} &(ref\langle c\rangle, t_1, \ldots, t_n) \rightarrow t_r \\ &(ref\langle c\rangle, t_1, \ldots, t_n) \rightarrow ref\langle c\rangle \\ &(ref\langle c\rangle) \rightarrow ref\langle c\rangle \end{aligned}$$

respectively.

Constructors, destructors and member functions are created anywhere within the IR tree by implementing its behavior using a lambda expression ((func) of Definition 3.4) and typing it using a corresponding type. In case a virtual member function should be invoked, a literal with the corresponding name has to be created and typed using a member-function type. The implementation then has to be present within some *ClassMetaInfo* container attached to the targeted class type or one of its sub-types as it is covered next.

ClassMetaInfo The structure designed to represent classes does not cover the implementation of named (virtual) member functions or essential constructors or destructors as they might be required by external libraries (see the requirement regarding *concepts* above). This information is stored within an annotation attached to the class type – the *ClassMetaInfo*.

Definition 3.59 (class meta info)**.** Let $\mathbb{L} \subset \mathbb{E}$ be the set of lambda expressions, $c_1, \ldots, c_n \in \mathbb{L}$ be lambdas of a constructor type, $d \in \mathbb{L}$ be a lambdas of a destructor type, $v, v_1, \ldots, v_m, k_1, \ldots, k_m \in \mathbb{B}$ be boolean flags, $n_1, \ldots, n_m \in \mathbb{I}$ be names and $f_1, \ldots, f_m \in \mathbb{L}$ be lambda expressions of a member function type. Then

$$\begin{array}{l}\text{ClassMetaInfo } \{\\ \quad c_1, \ldots, c_n\\ \quad v\ d\\ \quad n_1 : v_1\ k_1\ f_1, \ldots, n_m : v_m\ k_m\ f_m\\ \}\end{array}$$

is an instance of a ClassMetaInfo object aggregating function implementations associated to a class type. All included constructors, destructors and member functions have to be based on the same class type c. An instance of this annotation may only be attached to type c.

The constructors $c_1, \ldots, c_n$ are methods for constructing instances of type c and might include default, copy and move constructors. The destructor d defines the function to be invoked for destructing an object and the flag v determines whether the destructor is a virtual function. Finally, the list $n_1 : v_1\ k_1\ f_1, \ldots, n_m : v_m\ k_m\ f_m$ enumerates all named $(n_1, \ldots, n_m)$ member functions $f_1, \ldots, f_m$ to be attached to the class. Those member functions may be virtual $(v_1, \ldots, v_m)$ and/or constant $(k_1, \ldots, k_m)$. In the latter case they are not allowed to the directly or indirectly modify fields of the object passed as the first argument (*this* pointer).

C++ References As has been covered in the requirement section, references are essentially syntactic sugar for pointer based operations. Hence, internally C++ references are converted utilizing similar primitives. Pointers are covered by IR references (see Section 3.8.2). Consequently C++ references are encoded utilizing IR references as well. However, to provide means to distinguish pointers from references – which is required when passing references / pointers to external functions, C++ references are encoded by wrapping the corresponding IR reference into a struct. For instance, a C++ reference **int**& is encoded within INSPIRE using something similar to

```
struct { cpp_ref : ref<int<4>> }
```

Corresponding generic, derived packing and unpacking operations like

```
(struct { cpp_ref : ref<α> } r) → ref<α> {
  return r.cpp_ref;
}
```

to bridge the gap between IR references and encoded C++ references are incorporated as required. As will be shown in the following chapter, these kind of wrapper operations can be implicitly processed by our analysis framework. Similar approaches have been followed for encoding const modifiers and enumerations.

Expressions

Unlike for the type system no expression constructs need to be extended or added. The existing infrastructure for the definition of lambdas combined with the extended set of type constructors is sufficient to express constructors, destructors and member functions in addition to the pre-existing support for ordinary functions. Corresponding calls can as usually be conducted utilizing call expressions (rule (call) of Definition 3.4).

Virtual Functions However, although the syntax is not modified, the semantic of call expressions is extended to support the resolution of virtual member functions. To invoke a virtual member function a literal matching the corresponding name and type has to be targeted by a function call.

Example 3.24 (virtual function call)**.** Let A be a class type annotated with a ClassMetaInfo object including an entry for a virtual member function f of type $A :: (int\,\langle 4\rangle) \rightarrow unit$. Further, let B be a class derived from A. To invoke the virtual member function f on an object x of the dynamic type $ref\,\langle B\rangle$ a call to a literal

$$f \;:\; A :: (int\,\langle 4\rangle) \rightarrow unit$$

is used. Such a call to a member function literal triggers the implicit virtual function resolution. Hence, the call expression is searching within the meta information attached to B for an implementation of f of the corresponding type. If found, it is processed, otherwise the search continues within A's meta information.

For comparison, in case a specific implementation of f shall be invoked, thereby skipping the resolution of the virtual function, the literal representing the target of the call expression has to be replaced by the lambda expression encoding the corresponding implementation.

Member Field Access Another minor extension has to be added to provide sub-classes access to members of their parent types. Typically members of a struct are accessed utilizing the sub-referencing extension (see Section 3.8.2). This mechanism provides means for constructing data paths from the root element (the full object) to the sub-elements to be addressed. Since fields may be contained within parent classes an additional data path constructor providing the possibility to navigate to a base class of a given reference has to be added. Therefore the abstract operator

$$dp.parent : (datapath, type\,\langle\alpha\rangle) \rightarrow datapath$$

is added. In combination with the $ref.narrow$ and $ref.expand$ operators this construct supports the necessary navigation step from a derived class to one of its base classes as well as in the other direction.

Statements

No additional statements have to be added to cover the object-oriented aspects of C++. However, the semantic of the compound statement has to be updated and two new constructs supporting the raising and handling of exceptions are required.

Compound Statement Modification Every compound statement is defining a scope for the life-time of memory locations on the stack. Consequently, at the end of the scope all those memory locations need to be freed, as it is already the case for the basic IR. For the C++ support this responsibility is extended by the requirement for the compound statement to call the destructor for class-values in the reverse order of their allocation before releasing their storage space. While in the current implementation the destruction of objects is considered implicit, explicit destructor calls should be added at the end of scopes in a future development step.

The *throw* Statement Let $e \in \mathbb{E}$ be an arbitrary expression of type $t \in \mathbb{T}$. The statement

$$throw\ e$$

is a statement interrupting the current control flow and continuing execution in the most closely nested enclosing scope of a $try \ldots catch$ statement capable of handling values of type t.

The *try . . . catch* Statement Let $s, h_1, \ldots, h_n \in \mathbb{S}$ be a statement and $v_1, \ldots, v_n \in \mathbb{V}$ be variables. The statement

```
try
  s
catch(v_1) h_1
...
catch(v_n) h_n
```

is establishing a try-catch scope. Any exception e thrown / raised within statement block s will be check against the types of the variables $v_1, \ldots, v_n$.

If any of those types is a super-type of the type of e or of the abstract type *any* the corresponding exception handler routine $h_1, \ldots, h_n$ is processed and the program continues after the try-catch scope. In case non is matching, the exception is forwarded to the next enclosing try-catch scope.

The variable v_i declared within the *catch* expression is visible within the corresponding handler function h_i except if it is of type *any*. Variables of the *any* type may be used to encode the C++ catch-all construct *catch*(...).

Exceptions may not be thrown across the boundaries of $jobs$, work sharing constructs ($pfor$) or the data distribution primitive (*redistribute*). Doing so will result in undefined behavior. Correspondingly, exceptions may only be thrown and caught within a single thread of execution.

3.11.3 Modeling C++ Constructs

To conclude this brief overview on the support of C++ constructs within the Insieme IR examples demonstrating the encoding of essential language features shall be outlined.

A Simple Class Hierarchy

Consider the following C++ class definition covering a variety of the features incorporated into the Insieme IR.

```
struct A {
  int x;

  // a constructor
  A(int x) : x(x) {}

  // a non-virtual member function
  void f(int a) { ... };

  // a virtual member function
  virtual int g() { .. };
}

struct B : public A {
  int y;

  // a constructor
  B(int x, int y) : A(x), y(y) {}

  // implementation of A's virtual g for type B
  int g() { ... };
}
```

Class A is defined to contain a single integer and class B extending it by an additional member field. The corresponding encoding of those types in INSPIRE is similar to

```
struct {
  x : int<4>;
}
```

and

```
struct : struct { x : int<4>; } {
  y : int<4>;
}
```

Let A denote the first type and B the second. The encoding of the constructor of class A would be of type $A :: (int\langle 4\rangle)$ and similar to

```
(ref<A> this, int<4> x)→ref<A> {
  (*this).x := x;
  return this;
}
```

Let *A_ctor* be the encoding of A's constructor. Than B's constructor of type $B :: (int\langle 4\rangle, int\langle 4\rangle)$ is represented by

```
(ref<B> this, int<4> x, int<4> y)→ref<B> {
  A_ctor(this,x);
  (*this).y := y;
  return this;
}
```

Hence, C++'s (implicit) constructor initializer lists are explicitly represented by nested constructor calls and assignments. Also the forwarding of the implicit *this* pointer is made explicit by the first parameter.

The creation of objects on the stack or on the heap conducted by

```
A a1(12);
A* a2 = new A(14);
```

is encoded by

```
ref<A> a1 = A_ctor(ref.var(A),12);
ref<A> a2 = A_ctor(ref.new(A),14);
```

unifying the treatement of heap and stack allocated memory.

The non-virtual member function f of class A is encoded by

```
(ref<A> this, int<4> a)→unit {
  ...
}
```

and exhibits the type $A :: (int\langle 4\rangle) \rightarrow unit$. A corresponding member function call on an object conducted by the C++ code snippet

```
A a(1);
a.f(2);
```

is encoded as

```
ref<A> a = A_ctor(ref.var(A),1);
(ref<A> this, int<4> a)→unit {
  ...
}(a,2);
```

which is just the same as any other function call when interpreting f's type as $(ref\langle A\rangle, int\langle 4\rangle) \to unit$. The same function can be applied to objects of type B since B is a subtype of A and may hence be passed to f as its first argument.

Finally, let gA and gB represent the IR versions of the two virtual functions g of class A and B respectively. Hence, gA is bound to the name g within A's meta class information while within B's meta info g is mapped to gB. To invoke the virtual function a call expression targeting a literal defining the name of the function and its type has to be created. In the given case this literal is

$$g : A :: () \to int\langle 4\rangle$$

and a virtual member function call within C++ like

```
A& r = < some source > ;
r.g();
```

is represented in our IR by

```
ref<A> r = < some source > ;
g(r);
```

This will trigger the virtual function lookup operation which is searching for the corresponding implementation of g based on the dynamic type of r. If the call shall be statically bound to A's implementation the encoding would be

```
ref<A> r = < some source > ;
gA(r);
```

where gA is the lambda defining A's version of the member function g.

External Template Libraries

As has been covered in the requirements section, among the most important requirements on the design of the C++ additions has been the support of external generic libraries. We will quickly demonstrate our solutions ability of dealing with such based an the frequently utilized $std::vector$ container of the *Standard Template Library*.

The generic C++ type $std::vector\langle T\rangle$ is encoded in our IR using the abstract (generic) type $std::vector\langle\alpha\rangle$. However, like in C++, actual instances require a proper type variable instantiation. Let us consider a simple example creating a vector of integers in C++. The corresponding C++ code fragment is given by

```
std::vector<int> v;
```

which is creating a new vector value on the local stack. It does so by invoking a constructor, which we have to cover in our IR encoding. However, since the operation shall be kept abstract since we are interfacing a third-party library we do not utilize a lambda based implementation for the constructor. Instead we use an abstract constructor literal

$$vector\_ctor : std :: vector \langle\alpha\rangle :: (type \langle\alpha\rangle)$$

for the initialization. Note that meta-type parameters are ignored by the code generation yet necessary for the IR type system. The corresponding IR code would therefore look like

```
ref<std::vector<int<4>>> v =
  std::vector(            // the ctor-literal
    ref.var(std::vector<int<4>>),
    int<4>                // the type parameter
  );
```

Support for operators to be applied on the vector are added correspondingly. For instance, in C++ a new element can be added to the end of the list represented by a vector instance by

```
v.push_back(5);
```

To encode this operation into IR a generic abstract literal

$$push\_back : std :: vector \langle\alpha\rangle :: (\alpha) \rightarrow unit$$

representing the templated *push_back* operation is introduced and utilized. The corresponding encoded code fragment would therefor be equivalent to

```
push_back(v,5);
```

Hence the generic type system of or IR is utilized for modeling equivalent features of C++ templates. Furthermore, as for abstract generic types introduced by language extensions, single literals are capable of describing a full family of abstract operators. For instance, the *push_back* literal above is covering this abstract operation for any instantiation of the generic C++ vector class. This way of addressing families of generic operator instances utilizing a single construct can e.g. be utilized for simplifying the specification and/or implementation of analysis since the effects of a few generic primitives may be specified instead of a variety of individual instances.

3.12 Summary

In this chapter the novel design of the Insieme intermediate representation has been covered. A full description of the syntax and semantic of the included type, expression and statement constructs has been provided. The resulting holistic, parallelism aware, high-level intermediate representation is novel in the are of general purpose compilers targeting parallel programs [48]. The preservation of the high-level structure, the enforcement of a holistic view on the processed program, its concise design, its unified parallel model, and the utilization of abstract data types to focus on relevant aspects of a processed code makes the resulting IR an unprecedented foundation for static analyses and manipulations of parallel codes – as will be further investigated by the following chapters. Furthermore, its concise nature facilitated the complete formal specification of its semantic, as has been conducted in Section 3.7. This specification provides the foundation for reasoning about programs encoded utilizing the Insieme IR as well as for the development of comprehensive and advanced program analysis infrastructures (see Chapter 4). In turn, this capability of reasoning about (parallel) programs enables static optimizers to conduct save code transformations to tune a processed program to improve arbitrary objectives including the reduction of its execution time, the improvement of its parallel efficiency or its scalability (see Chapter 6).

Chapter 4

Analyses

Besides their purpose of serving as a common format bridging the gap between collections of input and output languages, compiler IRs of optimizing compilers also provide the foundation for analyses and transformations to be employed for improving the quality of processed programs. The former – the analysis of codes – is the topic of this chapter, while transformations will be covered in the following Chapter 5. The strategic application of those, to improve the quality of codes, is the topic of Chapter 6.

The area of program analyses covers a wide spectrum. Simple techniques range from basic primitives provided for navigating an IR, over the implementation of a tool box of basic IR inspection utilities, to flow-insensitive analyses including type checks and the extraction of code features to characterize codes. The latter may, for instance, be utilized for *machine learning* based optimization approaches. More elaborate techniques range from flow- and (potentially) context-sensitive analyses considering the actual (interprocedural and parallel) control flow of a program, over sophisticated approaches like e.g. polyhedral model based analyses, to dynamic program analyses involving the observation of actual executions of the processed code. All of those are – to a certain extend – supported by the Insieme compiler infrastructure and covered in this chapter.

This chapter starts by a brief overview on the means offered by the Insieme infrastructure for navigating its IR in Section 4.2, followed by an overview on flow-insensitive analyses in Section 4.3. The main contribution of this chapter, however, is a flow- and context-sensitive *constraint based analysis* (CBA) framework covering all INSPIRE language constructs, including its parallel primitives, its functional core encompassing functions and closures as first-class citizens and the related *dynamic dispatch problem*. Furthermore, the majority of its extensions, in particular including the mutable state extension, are supported. This framework is covered in Section 4.4. Finally, Section 4.5 and 4.6 provide a brief overview on the support and integration of polyhedral model based and dynamic analyses.

4.1 Contributions

The major contributions of this chapter are:

- the establishment of an infrastructure for flow-insensitive analyses processing the Insieme IR, including a feature extraction framework for characterizing codes (Section 4.3)
- the development of a novel, comprehensive, flow- and context-sensitive constraint based analysis framework for programs encoded utilizing the Insieme IR offering analytical capabilities at an unprecedented scale for general purpose compilers targeting parallel codes (Section 4.4)
- the demonstration of the utilization of the beneficial traits of our IR structure, in particular its high-level nature, for conducting polyhedral model based analyses (Section 4.5)
- the facilitation of dynamic analyses targeting non-functional properties of programs observed during their execution (e.g. execution time or energy consumption) by utilizing the close coupling between the compiler and runtime system (Section 4.6)

In the context of this thesis, this chapter demonstrates that the IR developed in the previous chapter, following the criteria of the thesis's hypothesis, provides a valuable foundation for conducting analyses targeting higher-level concepts, in particular including issues related to coarse grained parallelism and the tuning of (parallel) programs.

4.2 Navigating the IR

The foundation of all the analyses presented in this chapter and implemented in the Insieme infrastructure is provided by the basic facilities for navigating IR data structures, which are thus to be covered first.

Basic Steps

As has been covered in the implementation Section 3.10.2, nodes in the IR tree – which is physically realized as a DAG – can be addressed by *node pointers* referencing individual node instances or *node addresses* covering additional context information. Both addressing modes are fully typed. Hence, for instance, a reference to an IR node representing a call expression may be addressed by a *call expression pointer* or by one of its super types – an *expression pointer*, a *statement pointer* (since every expression is a statement), or, in the most general case, a *node pointer*. Correspondingly a *call expression address*, an *expression address*, a *statement address* or a *node address* may be utilized in case the path to a call expression node shall be denoted. Naturally, means for identifying the type of a referenced node are offered as well.

Each object referencing a node – either pointer or address based – provides typed access to named sub-structures. For instance, a reference to a call expression provides member functions to access its sub-structures, including the expression representing the function to be invoked as well as the involved arguments. The result of accessing an argument of a *call expression pointer* is a *expression pointer* while the same operation been applied on a *call expression address* yields an *expression address.* In the latter case, the resulting address is an extended version of the path referencing the original call expression node.

When referencing IR nodes utilizing node addresses, the parent node of the addressed node in its current logical context can be obtained, unless the path is only consisting of a single root node, while for node pointers an upward navigation in the IR tree can not be provided.

For a variety of small inspection operations those basic navigation facilities are already sufficient. For instance, testing whether a given type is representing an array type and extracting the element type or similar simple, yet ubiquitously required operations can be easily implemented utilizing those facilities.

In addition to named sub-structures, every node reference provides access to the full list of sub-structures to realize support for generic visitor operations. However a direct utilization of those is discouraged in favor of a collection of higher-order visitor functions providing effective generic implementations of a variety of IR tree traversal orders.

Visitors

Logically the IR data structure is a tree and most algorithms based on it require support for traversing it. Therefore a set of higher-order functions offering type save IR tree traversal strategies are provided.

The two basic tree traversal strategies are the well known *depth-first* and *breath-first* strategies where the first may be conducted in *pre-* or *post-order.* Additionally a *parallel* visiting strategy is provided, where the involved nodes are visited in an arbitrary order, yet simultaneously by multiple concurrent threads. Those strategies provide the foundation of the implemented higher-order tree traversal functions. Additionally each traversal may be customized by an orthogonal set of optional modifiers. Those options include:

- *Visit-Once Option:* with this option a visitor is visiting every instance of a node within the IR DAG only once. For instance, a visitor traversing the term $A(B(C), B(C))$ in depth-first post-order would visit the node sequence C, B, C, B, A while in depth-first post-order with the visit once option the sequence C, B, A would be visited. The second B and its sub-nodes are skipped. Due to the high amount of node sharing – in particular for type constructs and literals – this option can significantly reduce the number of visited nodes and hence the run-time complexity of algorithms if utilized properly.

- *Interruptible Option:* with this option a visitor is allowed to interrupt the tree traversal at any point during the traversal. Especially for implementing functions searching for sub-structures satisfying a given property this feature provides a convenient facility.
- *Prunable Option:* in some cases during the traversal of an IR tree the traversal of the sub-structures of a given node can be skipped since the effect of their traversal is known. For instance, when searching for a given expression type constructs and all their sub-structures within the IR tree may be skipped since it is known that an expression can never be a sub-structure of a type. Such a pruning is supported by this option. At every step visitors are asked whether sub-structures shall be covered or may be skipped.
- *Node Pointer or Node Address Based:* the traversal of the tree may be conducted utilizing node pointers or node addresses. The selection of which kind of addressing mode is utilized can be freely chosen. While pointers are fast, node addresses provide the sometimes required capability of navigating the current context of a visited node – in particular its parent node could be obtained.
- *Filters:* in many cases algorithms are not interested in every node within an IR tree. Filters may therefor be specified to select a sub-set of the nodes to be visited. A convenient one, for instance, is to filter out a certain set of node types. This way visitors only visiting e.g. for loops and checking those for certain requirements may be realized.

The implementation of those tree traversal operations is based on C++ templates and the interface is utilizing C++11 lambdas for the convenience of the developer formulating algorithms on top of them. For instance, to check whether a given code fragment code contains e.g. a for-loop, the corresponding operation can be realized by

```
bool res = visitDepthFirstOnceInterruptible(
  node, [](ForStmtPtr cur) { return true; }
);
```

where visitDepthFirstOnceInterruptible is the higher-order visitor function realizing the interruptable depth-first visiting where every node is only visited once and the C++11 lambda

```
[](ForStmtPtr cur) { return true; }
```

the filter identifying the for loop statements to be looking for. By returning true the tree traversal is interrupted at the first encounter of a for loop and visitDepthFirstOnceInterruptible returns true if an interrupt occurred. The decision to use node pointers for the tree traversals and to filter out any node not being a for statement is implicitly deduced by the signature of the filter utilizing C++ template meta programming facilities.

4.3 Flow-Insensitive Analyses

Although limited in their accuracy, flow-insensitive analysis provide fast means for deducing static properties of program codes. Besides others, any property that can be deduced from an IR tree by induction over its structure is the result of a flow-insensitive analysis and may therefore be implemented as such. The required infrastructure to do so is provided by the basic IR navigation and visitor infrastructure presented in the previous section.

The main purpose of this section, however, is to demonstrate the utilization of the preservation of higher level constructs in the Insieme IR to implement simple, flow-insensitive analyses applicable in the context of a variety of operations. Those include the verification of the proper composition of language constructs, the extraction of static code features, e.g. for characterizing codes for machine learning based approaches, and a variety of simple code transformations. Examples of such analyses and their application will be outlined in the following sub-sections.

4.3.1 Type Checks and Validity Constraints

While of limited relevance for actual code optimization passes, the support of an automated utility verifying the proper composition of IR code fragments is among the most productivity increasing features regarding the implementation of IR utilities. In particular the development of IR frontends, new IR extensions involving the definition of derived operators, or the implementation of code transformations can benefit from such an IR-check infrastructure capable of validating produced IR codes.

Essentially the corresponding operations implement the type checking procedure of Definition 3.31 and the validity constraints introduced in Section 3.6. All of these are based on IR tree traversals conducting local checks on the encountered nodes and the collection of identified issues to be reported to the user.

4.3.2 Code Features

Another application of flow-insensitive analysis is the extraction of code features from a given program fragment. Support for this kind of operations is, for instance, required when applying machine learning based optimization strategies on program codes [58]. In such a setup code features are extracted from code fragments to characterize their behavior. Such features may include simple counters covering the number of integer or floating point operations, memory accesses, ratios between instruction types to describe the instruction mix, the number of branches, loops or function calls, the maximum loop nesting depth or the presence of recursive function calls. All these values can be retrieved directly by a simple traversal of the IR tree.

A Simple Feature Extraction Framework

The extraction of static program features can be formalized in a generic framework. A feature is thereby defined three components:

- a *value set* $\mathcal{V}$
- an *extractor function* ϵ and
- an *aggregation function* $\sqcup$

The value set is defining the domain of the represented feature, the extractor function is extracting a corresponding value from a given IR node and the aggregation function defines how the values of sub-structures have to be aggregated to compute the feature value of a composed construct.

A simple feature extraction framework may just count the number of static occurrences of nodes exhibiting properties of interest, like the invocation of a class of operators, within a given code fragment. This operation is realized by the framework covered in the following definition.

Definition 4.1 (static feature extraction framework)**.** Let $\mathbb{IR} = \mathbb{T} \cup \mathbb{E} \cup \mathbb{S}$ be the set of all IR nodes. A feature f is defined by a triple $f = (\mathcal{V}, \epsilon, \sqcup)$ where $\mathcal{V}$ is an arbitrary domain, ϵ is a function of type $\mathbb{IR} \to \mathcal{V}$ and $\sqcup$ a function of type $\mathcal{V}^* \to \mathcal{V}$. The feature value $\nu_f(n)$ of an IR node $n \in \mathbb{IR}$ is computed by the function $\nu_f : \mathbb{IR} \to \mathcal{V}$ defined by

$$\nu_f(n) = \epsilon(n) + \bigsqcup_{i=1}^{|n|} (\nu_f(n[i]))$$

where $|n|$ determines the number of child nodes of a node n and $n[i]$ is the i-th child node of n.

This simple formalism is sufficient to count the number of static instructions present in programs as it is utilized for characterizing loops and other code fragments in machine learning based optimizing compilers [58, 63]. The following example outlines the required instantiation of the involved components.

Example 4.1 (counting operations)**.** Let $OP \subset \mathbb{E}$ be a set of abstract or derived operators. To describe the static number of applications of operators in OP within a given code fragment a feature

$$c_{OP} = (\mathbb{N}, \epsilon_{OP}, \sqcup_{OP})$$

can be defined where $\epsilon_{OP} : \mathbb{IR} \to \mathbb{N}$ is given by

$$\epsilon_{OP}(n) = \begin{cases} 1 & \text{if } n = f(a_1, \dots, a_n) \text{ and } f \in OP \\ 0 & \text{otherwise} \end{cases}$$

and $\sqcup_{OP} : \mathbb{N}^* \to \mathbb{N}$ is defined by

$$\sqcup_{OP}([f_1, \dots, f_n]) = \sum_{i=1}^{n} f_i$$

Thus, the extractor ϵ_{OP} is identifying calls of operators in OP and the aggregation function $\sqcup_{OP}$ is summing up the number of occurrences encountered in sub-structures. Combined with the framework introduced by Definition 4.1 a value for a feature $c_{\{+\}}$ is extracted from a code fragment $n \in \mathbb{IR}$ by evaluating $\nu_{c_{\{+\}}}(n)$. For instance, applied to a simple code fragment n equal to

```
s := *s + *a[i];
```

the evaluation of $\nu_{c_{\{+\}}}(n)$ yields 1, while $\nu_{c_{\{*\}}}(n)$ yields 2 and $\nu_{c_{\{+,*,:=\}}}(n)$ is equivalent to 4.

Similar to operations, the number of loops, conditional statements, function calls or recursive function calls may be counted by customizing the extractor function ϵ accordingly. Furthermore, the setup can be utilized to compute features including the maximal depth of contained loop nests.

Example 4.2 (maximum loop nesting depth)**.** To extract the maximum number of nested loops a feature

$$(\mathbb{N}, \epsilon_{nl}, \sqcup_{nl})$$

where $\epsilon_{nl} : \mathbb{IR} \to \mathbb{N}$ is given by

$$\epsilon_{nl}(n) = \begin{cases} 1 & \text{if n = for (\dots) \dots} \\ 1 & \text{if n = while (\dots) \dots} \\ 0 & \text{otherwise} \end{cases}$$

and the aggregation function $\sqcup_{nl} : \mathbb{N}^* \to \mathbb{N}$ is given by

$$\sqcup_{nl}([f_1, \dots, f_n]) = \max_{i=1}^{n} f_i$$

can be utilized.

Furthermore derived features like the ratio between operators or vectors of features can be realized utilizing the same infrastructure.

Example 4.3 (instruction ratios)**.** Let $OP_1, OP_2 \subset \mathbb{IR}$ be two sets of operators for which the ratio shall be obtained (e.g. arithmetic operations vs. load/store operations or floating point operations vs. all operations). The corresponding feature

$$(\mathbb{R}, \epsilon_r, \sqcup_r)$$

is given by $\epsilon_r : \mathbb{IR} \to \mathbb{R}$ defined by

$$\epsilon_r(n) = \nu_{c_{OP_1}}(n)/\nu_{c_{OP_2}}(n)$$

and the aggregation function $\sqcup_r : \mathbb{R}^* \to \mathbb{R}$ is given by

$$\sqcup_r([f_1, \dots, f_n]) = 0$$

This definition is effectively ignoring the recursive part of the feature extractor and simply utilizing the definition of the operator-counting features presented above.

Vectors of features are realized by combining the domains of the individual features correspondingly and applying the extraction and aggregation functions element wise on the resulting, combined feature vector.

Weighted Features Although establishing a foundation for the extraction of static code features by simply counting and aggregating the presence of constructs, the basic framework of Definition 4.1 does not consider the context of a processed node. For instance, when obtaining the feature $c_{\{+\}}$ from the code fragment

```
ref<int<4>> s = var(0);
for(int<4> i = 0 .. size : 1) {
  s := *s + *a[i];
}
```

the result is 1 – since there is only one “static” instance of the + operator. However, in particular when aiming for characterizing the instruction mix of a code fragment, instructions within loops and recursions should have a higher weighting as instructions outside. Also, instructions within branches of a conditional statement (if) should have a reduced weight since their likelihood of being executed is less than 1 in the general case.

We therefore extended the framework presented above by the capability of weighting branches of an execution. Therefore the type of the *aggregation function* $\sqcup$ is extended to $\sqcup^+$ of type $(\mathbb{R} \times \mathcal{V})^* \to \mathcal{V}$ where each element of the input set consists of a weight and the value obtained for one of the substructures of the currently evaluated node. The corresponding framework extension is covered within the following definition.

Definition 4.2 (weighted feature extraction framework)**.** Let $\mathbb{IR} = \mathbb{T} \cup \mathbb{E} \cup \mathbb{S}$ be the set of all IR nodes. A feature f is defined by a triple $f = (\mathcal{V}, \epsilon, \sqcup^+)$ where $\mathcal{V}$ is an arbitrary domain, ϵ is a function of type $\mathbb{IR} \to \mathcal{V}$ and $\sqcup^+$ a function of type $(\mathbb{R} \times \mathcal{V})^* \to \mathcal{V}$. Let $a \sqcup^+ b$ be equivalent to $\sqcup^+([a, b])$ for any $a, b \in \mathbb{R} \times \mathcal{V}$. The feature value $\nu_f^+(n)$ of an IR node $n \in \mathbb{IR}$ is computed by the function $\nu_f^+ : \mathbb{IR} \to \mathcal{V}$ defined by

$$\nu_f^+(n) = \epsilon(n) + \begin{cases} \left(1, \nu_f^+(c)\right) \sqcup^+ \left(0.5, \nu_f^+(t)\right) \sqcup^+ \left(0.5, \nu_f^+(e)\right) & \\ \qquad \text{if } n = \text{if c then t else e} & \\ \left(1, \nu_f^+(x)\right) \sqcup^+ \left(1, \nu_f^+(y)\right) \sqcup^+ \left(1, \nu_f^+(z)\right) \sqcup^+ \left(w_l, \nu_f^+(b)\right) & \\ \qquad \text{if } n = \text{for(type i = x .. y : z) b} & \\ \left(w_l, \nu_f^+(c)\right) \sqcup^+ \left(w_l, \nu_f^+(b)\right) & \\ \qquad \text{if } n = \text{while c do b} & \\ (1, \nu_f^+(g)) \sqcup^+ \bigsqcup^+_{1 \le i \le n} \left(1, \nu_f^+(a_i)\right) & \\ \qquad \text{if } n = g(a_1, \ldots, a_n) \text{ and } g \text{ is a non-rec. lambda} & \\ (w_r, \nu_f^+(g)) \sqcup^+ \bigsqcup^+_{1 \le i \le n} \left(1, \nu_f^+(a_i)\right) & \\ \qquad \text{if } n = g(a_1, \ldots, a_n) \text{ and } g \text{ is a rec. lambda} & \\ \bigsqcup^{+|n|}_{i=1} \left(1, \nu_f^+(n[i])\right) & \\ \qquad \text{otherwise} & \end{cases}$$

where $|n|$ determines the number of child nodes of a node n and $n[i]$ is the i-th child node of n. The parameters w_l and w_r provide means for adjusting the weighting of loop iterations and recursive functions respectively.

For the the weighted framework, the aggregation functions presented above need to be adapted accordingly. For the operator-counting features the aggregation function $\sqcup_{OP}$ has to be updated to

$$\sqcup^+_{OP}([(w_1, f_1), \ldots, (w_n, f_n)]) = \sum_{i=1}^{n} w_i f_i$$

such that weights are considered properly. The aggregation function of the maximum loop-nest depth features, however, is defined by

$$\sqcup^+_{nl}([(w_1, f_1), \ldots, (w_n, f_n)]) = \max_{i=1}^{n} f_n$$

ignoring the weights since loop iterations are not effecting the nesting levels of loops and by defining $\sqcup^+_r$ by

$$\sqcup^+_r([(w_1, f_1), \ldots, (w_n, f_n)]) = 0$$

the features obtained from sub-structures remain ignored when extracting instruction ratios. However, ϵ_r has to be updated to

$$\epsilon_r^+(n) = \nu_{c_{OP_1}}^+(n)/\nu_{c_{OP_2}}^+(n)$$

to benefit from the weighted feature extraction framework.

The effect of the weight extension is best illustrated by an example. For instance, consider the code fragment n representing

```
s := x + y;
for(int<4> i = 0 .. size : 1) {
   if (p(i)) {
      s := *s + *a[i];
   }
}
```

The evaluation of the feature $c_{\{:=\}}$ by computing $\nu_{c_{\{:=\}}}(n)$ will yield 2 – since there are 2 assignments included in the representation – while $\nu_{c_{\{:=\}}}^+(n)$ will yield 51, assuming the parameter $w_l = 100$. This is due to the fact that the if branch of the body is considered to be processed 50% of the times an the loop is expected to be iterated 100 times. Both are assumptions made by the framework which are inherently required in the one or the other form due to the lack of information regarding the actual behavior of loops and conditional statements.

The weighted feature extraction has been shown to more accurately capture the characteristic of code fragments and has been utilized, for instance, for tuning the execution of OpenCL kernels based on our infrastructure [56].

Limitations Naturally, static program features like the number of arithmetic instructions or load/store operations are mere approximations of the actual number of instructions issued while processing the corresponding code fragment. Various restrictions on the available information, in particular regarding the actual control flow, result in imprecise estimations. Also, dynamically dispatched function calls, as they can occur when utilizing function pointers in C or virtual member functions in C++, can not be handled by this simple approach. Furthermore, code transformations like loop unrolling or tiling will have a high impact on the estimated number of processed operations, although the actual number of issued instructions is not affected. However, relative metrics, like the ratio between arithmetic operations and load/store instructions, are more resilient. Nevertheless, as for all feature based categorization approaches, the features utilized for the deduction of properties of program codes have to be carefully selected and statistically tested for their actual significance.

4.3.3 Local Transformations

A third application for simple, flow-invariant analyses in the Insieme compiler infrastructure are a variety of simple code transformations and manipulation utilities build upon on those. Two categories of these operations shall be briefly outlined at this point.

Constant Folding and Propagation

The first category of those utilities focus on pattern based code re-writing operations. Hence, on situations in which the structure of a local code fragment is inspected and, if it fits a certain pattern, restructured into an equivalent, yet preferable shape. Example transformations of this kind are local constant folding, constant propagation and dead code elimination.

Such operations can, for instance, reduce expressions like `1+2*3` to `7` and eliminating combinations of operators annihilating each other, like in the expression `ref.deref(ref.var(10))` which is equivalent to `10`. A more complex reductions is the elimination of a call to a locally created closure similar to

```
((int<4> y) => (*a + 12 + y)) (2*z)
```

by substituting it by the equivalent

```
*a + 12 + 2*z
```

eliminating the overhead of creating and invoking a closure. Naturally, this principle can be extended to statements. For instance, loops which will never be entered or unreachable branches of conditional statements may be eliminated too.

A more advanced operation is the simplification of types of variables. For instance, within the code fragment

```
{
  ref<int<4>> x = ref.var(10);
  for(int<4> i = 0 .. size : 1) {
    a[i] := *x + *a[i];
  }
}
```

the type of the variable *x* could be reduced to *int<4>* since after the initial assignment the state is never modified again. The analysis whether a variable is ever utilized as the target of an assignment operation can be conducted in a flow-insensitive manner utilizing a visitor. The type reduction produces

```
{
  int<4> x = 10;
  for(int<4> i = 0 .. size : 1) {
    a[i] := x + *a[i];
  }
}
```

which can be further simplified to

```
{
  for(int<4> i = 0 .. size : 1) {
    a[i] := 10 + *a[i];
  }
}
```

using another transformation of the "constant propagation" category exploiting the single-assignment characteristic of variables in our IR and the fact that the expression utilized for initializing x does not exhibit side effects – another property that can be determined by flow-insensitive analyses.

The application of each simplification step could enable the application of another simplification step. A collection of these kind of IR-simplification rules may therefore be iteratively applied on a given input code until no further step can be conducted. The infrastructure for defining such transformation rules is covered in more detail in Chapter 5, in particular within Section 5.3 and Section 5.5.

Code Manipulation Primitives

The second category of utilities are common manipulation primitives including utilities for inlining function calls or the reverse operation outlining statements into an isolated function. While the former may be utilized to eliminate function call overheads, the latter is e.g. frequently employed for implementing support for parallel APIs in the frontend. For instance, the body of a for loop needs to be outlined into a function before it can be utilized by a parallel for loop ($pfor$) in our IR. This outlining requires the collection of all free variables within the targeted code fragment, the construction of a lambda expression accepting those as parameters and a bind expression capturing the corresponding values from the local context. The necessary data can all be collected utilizing flow-insensitive analyses.

Other primitives based on flow-insensitive analyses include the unfolding and unrolling of recursive functions, which require the collection of all occurrences of recursive variables within the corresponding definitions, and the instantiation of generic IR types and functions (type variables).

4.4 Flow-Sensitive Analyses

In contrast to *flow-insensitive* analyses, *flow-sensitive* analyses consider the order of processed operations. For instance, while a flow-insensitive analysis may determine that two variables may (at any point) have the same value, a flow-sensitive analysis may determine that they may have the same value at a given point in the program's execution, e.g. before or after processing a specific statement.

Flow-sensitive analyses, in general, follow common patterns providing the opportunity of defining abstract frameworks to be utilized for their specification as well as for their implementation. The most widely known of those are conventional *data-flow analysis* (DFA) frameworks. However, others exist as well, exhibiting different traits [68]. One alternative approach, namely *constraint-based analysis* (CBA), offering an extended flexibility and additional capabilities is introduced in addition to the conventional DFA approach in the beginning of this section. The remainder of this section then describes the design, modification and integration of CBA based techniques for flow-sensitive analysis based on the Insieme intermediate representation, covering all its sequential and parallel constructs.

4.4.1 Overview on Flow-Sensitive Program Analysis

The purpose of static, flow-sensitive analysis is to deduce properties of programs at certain states of their execution. For instance, for a given code fragment, developers of code optimizations may be interested in the set of values still being alive, hence still to be utilized in a future program state, to identify dead code which can be eliminated without affecting the observable program behavior. Other analyses aim on value traits including e.g. the constancy of values, their sign or the set of memory locations a reference value may point to before or after processing a given statement.

For instance, in the code fragment

```
1  a := 0;
2  b := 1;
3  if (a < b) {
4    a := 4;
5    b := a + b;
6  }
7  return a;
```

one might be interested in whether some statements could be safely[1] removed. A *live variable analysis* – a classical data-flow analysis – would determine that the assignment of line 5 can be eliminated since the computed value is never read. A subsequent application of a *constant folding* analysis and a corresponding *constant propagation* transformation would yield

```
a := 0;
b := 1;
if (0 < 1) {
  a := 4;
}
return a;
```

which can be further simplified to

[1]safely = without altering the observable program behavior

```
a := 0;
b := 1;
a := 4;
return a;
```

which, with an additional round of a *live variable analysis* would result in

```
a := 4;
return a;
```

which, by utilizing *constant propagation* would finally be reduced to

```
return 4;
```

In the past, several techniques have been investigated and developed for static flow-sensitive analyses, including *data-flow analysis*, *constraint-based analysis* and *abstract interpretation* [68]. All of those analyzing techniques can be abstracted into frameworks to separate the description of the operation of the corresponding technique from the definition of actual analyses. Two of those shall be presented next – *data-flow analysis* (DFA) as a smooth, well known introduction and *constraint-based analysis* (CBA) as a advanced, more flexible approach which also provides the foundation for the following sections.

Data Flow Analysis

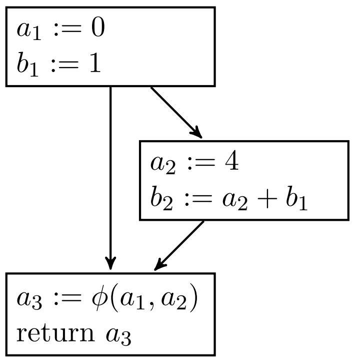
Data flow analyses are among the most well known approaches for static program analysis and descriptions can be found in any textbook covering compiler based program analyses. This brief summary shall therefore only serve as a foundation for the content to follow.

Data flow analyses are essentially solving a set of equations established over the structure of a *control flow graph*.

Definition 4.3 (control flow graph)**.** Let I be a set of low-level program instructions. A *basic block* $b \in I^*$ is a sequence of low-level instructions not including any label or jump instructions. A *control flow graph* (CFG) is a connected, directed graph $g = (B, E)$ where $B \in 2^{I^*}$ is a set of basic blocks and $E \subseteq B \times B$ a set of directed control flow edges.

An edge $(b_1, b_2) \in E$ in a CFG indicates that a program may continue its execution by processing the instruction sequence of block b_2 after finishing the processing of the instructions in b_1. The CFG therefore summarizes all potential flows of control through the represented program code.

Example 4.4 (control flow graph)**.** For instance, the CFG



summarizes the code fragment addressed by the motivating example above. The instructions in the basic blocks are encodes utilizing the *static single assignment* form (SSA) as it is frequently required by implementations to simplify analyses.

Many low-level compiler IRs are organized in the shape of a control-flow graph and may hence directly serve as the input of a corresponding data-flow framework implementation. The LLVM-IR, for instance, is additionally restricted to SSA form.

Before covering the actual data-flow analysis approach we have to provide one more definition introducing the structure utilized for modeling the properties to be deduced from a program.

Definition 4.4 (property space)**.** A *property space* is given by a pair

$$(L, \bigsqcup)$$

where L is a set of values the characterized property may exhibit and $\bigsqcup : 2^L \to L$ is a *combination operator.* Based on those, let the binary $\sqcup : L \times L \to L$ be defined by $l_1 \sqcup l_2 = \bigsqcup\{l_1, l_2\}$, the binary relation $\sqsubseteq: L \times L$ be defined by $l_1 \sqsubseteq l_2$ iff $l_1 \sqcup l_2 = l_2$, the binary relation $\sqsupseteq: L \times L$ be defined by $l_1 \sqsupseteq l_2$ iff $l_2 \sqsubseteq l_1$, $\bot = \bigsqcup \emptyset$ and $\top = \bigsqcup L$.

A property space consists of the set of values a property may exhibit plus an operator merging the results of multiple, optional paths. For instance, if a program may follow at some point two different paths, and the actual path is unknown, the property to be obtained is computed along both paths and combined utilizing the $\sqcup$ operator.

Example 4.5 (property space)**.** Let V be the set of variables encountered within the SSA-CFG to be analyzed. When applying a *live variables analysis* the corresponding property space is given by

$$(2^V, \bigcup)$$

Hence, every computed property is a subset of V and the union operator is utilized for combining values computed along alternative paths. The later is based on the fact that at any point in the program the set of variables to be read in the future corresponds to the union of the variables read along all the individual paths to be potentially followed from that point on. Also, we have $\sqsubseteq$ being the sub-set relation $\subseteq$ and the constants $\bot = \bigcup \emptyset = \emptyset \in 2^V$ and $\top = \bigcup 2^V = V \in 2^V$.

To the contrast, for a *constant folding* analysis focusing on a single integer variable the corresponding property space is given by

$$(C, \sqcup_c)$$

where $C = \mathbb{Z} \cup \{\bot_c, \top_c\}$ and $\sqcup_c : 2^C \to C$ is defined by

$$\sqcup_c(A) = \begin{cases} \top_c & \text{if } \top_c \in A \vee |A \cap \mathbb{Z}| > 1 \\ a & \text{if } \top_c \notin A \wedge (A \cap \mathbb{Z} = \{a\}) \\ \bot_c & \text{otherwise} \end{cases}$$

In particular we have $\bot = \sqcup_c \emptyset = \bot_c$ and $\top = \sqcup_c C = \top_c$. Over the course of the computation of the analysis result, the value of a variable might be $\bot_c$ which corresponds to 'not (yet) known but may still turn out to be a constant', $x \in \mathbb{Z}$ if it is the constant x or $\top_c$ if it has been established that the targeted variable is definitely not a constant.

Multiple property spaces can be combined to form new property spaces. For instance, let $(L_1, \sqcup_1)$ and $(L_2, \sqcup_2)$ be two property spaces. Than the pair $(L_1 \times L_2, \sqcup_{12})$ where $\sqcup_{12}$ is given by

$$\sqcup_{12}(\{(l_{11}, l_{21}), \dots, (l_{1n}, l_{2n})\}) = (\sqcup_1(\{l_{11}, \dots, l_{1n}\}), \sqcup_2(\{l_{21}, \dots, l_{2n}\}))$$

is a property space combining the two given spaces by forming pairs of values of their associated value sets. Similar component wise combinations may be realized for an arbitrary number of spaces. Also, let $(L, \sqcup)$ be another property space and K be an arbitrary set. Then the pair $(K \rightharpoonup L, \sqcup_{K \rightharpoonup L})$ consisting of the set of partial mappings between the key set K and the property domain L and the combination operator $\sqcup_{K \rightharpoonup L}$ defined by

$$\sqcup_{K \rightharpoonup L}(\{m_1, \dots, m_n\}) = \begin{cases} \epsilon & \text{if } n = 0 \\ m_1 \circ \sqcup_{K \rightharpoonup L}(\{m_2, \dots, m_n\}) & \text{otherwise} \end{cases}$$

where $\circ : ((K \rightharpoonup L) \times (K \rightharpoonup L)) \to (K \rightharpoonup L)$ is given by

$$m_1 \circ m_2 = \begin{cases} m_1 & \text{if } m_2 = \epsilon \\ (m_1 \circ m_2')[k \mapsto m_1[k] \sqcup l] & \text{if } m_2 = m_2'[k \mapsto l] \wedge k \in \text{dom}(m_1) \\ (m_1 \circ m_2')[k \mapsto l] & \text{if } m_2 = m_2'[k \mapsto l] \wedge k \notin \text{dom}(m_1) \end{cases}$$

is a property space as well. Utilizing those and other connectors, more complex property spaces can be created by combining simpler spaces. For instance, utilizing this connector, the property space for determining the constant value of a single variable covered above can be extended to a property space $(V \rightharpoonup C, \sqcup_{V \rightharpoonup C})$ covering all variables by utilizing the set V of variables as the key set. The obtained properties are then describing the constant values of all variables within the analyzed program.

In the literature, additional requirements on property spaces are discussed. For instance, frequently property spaces are required to be *complete lattices* or, to guarantee convergence of the data-flow algorithms, property spaces are required to satisfy the *Ascending Chain Condition* [68]. However, the corresponding details are of little relevance for this overview section and will be skipped for brevity.

The basic idea of data-flow analysis is to assign variables to all the entry and exit points of basic blocks of a control flow graph which are then utilized to formulate constraints among those. The domain of the variables is defined by a property space and the constraints are based on the combination operator and its derivatives. The following definition covers the general schema of data-flow analysis.

Definition 4.5 (data-flow analysis framework)**.** Let (B, E) be a control flow graph. A *data-flow analysis* is specified by a tuple

$$(L, \sqcup, \mathrm{t}, \{f, b\})$$

where $(L, \sqcup)$ is a property space, $\mathrm{t} : B \to (L \to L)$ is a family of *transfer functions* and $\{f, b\}$ determines whether it is a *forward* or *backward* analysis. Let t_b denote the transfer function $t(b)$. Further, let $\mathrm{pred} : B \to 2^B$ defined by

$$\mathrm{pred}(b) = \{x \in B \mid (x, b) \in E\}$$

be a function obtaining the predecessors $\mathrm{pred}(b)$ of a basic block $b \in B$ and $\mathrm{succ} : B \to 2^B$ defined by

$$\mathrm{succ}(b) = \{x \in B \mid (b, x) \in E\}$$

be a function obtaining the succeeding blocks. For all $b \in B$ let in_b and out_b be variables associated to the *in* and *out state* of the basic block b. The set of constraints on those variables is obtained by aggregating the constraints

$$\mathrm{in}_b \sqsupseteq \bigsqcup_{p \in \mathrm{pred}(b)} \mathrm{out}_p$$
$$\mathrm{out}_b \sqsupseteq t_b(\mathrm{in}_b)$$

for all $b \in \mathbb{B}$ for forward analysis or the constraints

$$\begin{aligned} \mathrm{in}_b &\sqsupseteq t_b(\mathrm{out}_b) \\ \mathrm{out}_b &\sqsupseteq \bigsqcup_{s \in \mathrm{succ}(b)} \mathrm{in}_s \end{aligned}$$

for backward data-flow analysis respectively. Based on those, an assignment for the variables in_x and out_x satisfying all the constraints is computed. Since their may be multiple valid assignments, the most restrictive assignment in terms of the $\sqsubseteq$ relation of the property space is to be chosen to obtain the most accurate results. This solution is also known as the *least fixpoint* of the given set of constraints. Let C be the set of constraints and $V = \bigcup_{b \in B} \{\mathrm{in}_b, \mathrm{out}_b\}$ the set of variables referenced by those constraints. A naïve algorithm to compute the desired least fixpoint assignment $A \in (V \rightharpoonup L)$ is given by the following pseudo code:

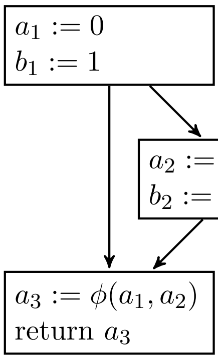
```
// Step 1: init A with the property space's $\perp$ value
$A := \epsilon$
for $b \in B$ do
    $A := A[\mathrm{in}_b \mapsto \perp]$
    $A := A[\mathrm{out}_b \mapsto \perp]$
end for

// Step 2: gradually fix unsatisfied constraints
while $\exists (v \sqsupseteq t) \in C \; . \; A[v] \not\sqsupseteq A(t)$ do
    $A := A[v \mapsto (A[v] \sqcup A(t))]$
end while
```

where $A(t)$ is the evaluation of the term t utilizing the assignment A. The algorithm exploits the shape of the constraints where in every case the left-hand side is a single variable.

As for *property spaces*, in literature a verity of additional properties on the transfer functions t are stated. In particular the restriction to monotone functions to guarantee the existence of a least fixpoint of the constraints is essential. However, those details are beyond the scope of this section and are therefore skipped for brevity. Interested readers may be referred to related literature for details regarding this subject [68].

In practical implementations the explicit generation of constraints is typically skipped due to their regular structure. The desired variable assignment is directly computed based on the CFG utilizing implicit constraints. However, we made them explicit as a preparation towards the concept of constraint-based analysis.

Example 4.6 (data-flow analysis)**.** Let us consider a *live variable analysis* on the control flow graph of Example 4.4 given by



For simplicity we strip of unnecessary details and obtain

$$\begin{tikzcd}
\boxed{\text{A}} \arrow[dr] \arrow[dd] & \\
 & \boxed{\text{B}} \arrow[dl] \\
\boxed{\text{C}} &
\end{tikzcd}$$

where the labels on the nodes correspond to the identifier we utilize for referencing to those basic blocks.

The *live-variable analysis* for the DFA framework is given by the tuple

$$(2^V, \cup, \mathrm{t}, b)$$

where $V = \{a_1, a_2, a_3, b_1, b_2\}$ and t is given by

$$\begin{aligned}
\mathrm{t}_A(X) &= X \setminus \{a_1, b_1\} \\
\mathrm{t}_B(X) &= (X \setminus \{a_2, b_2\}) \cup \{b_1\} \\
\mathrm{t}_C(X) &= (X \setminus \{a_3\}) \cup \{a_1, a_2\}
\end{aligned}$$

Since it is a backward analysis the framework will generate the following constraints based on the CFG:

$$\begin{aligned}
\mathrm{in}_A &\supseteq t_A(\mathrm{out}_A) = \mathrm{out}_A \setminus \{a_1, b_1\} \\
\mathrm{out}_A &\supseteq \mathrm{in}_B \cup \mathrm{in}_C \\
\mathrm{in}_B &\supseteq t_B(\mathrm{out}_B) = (\mathrm{out}_B \setminus \{a_2, b_2\}) \cup \{b_1\} \\
\mathrm{out}_B &\supseteq \mathrm{in}_C \\
\mathrm{in}_C &\supseteq t_C(\mathrm{out}_C) = (\mathrm{out}_C \setminus \{a_3\}) \cup \{a_1, a_2\} \\
\mathrm{out}_C &\supseteq \emptyset
\end{aligned}$$

where the generic $\sqcup$, $\sqsupseteq$ and $\perp$ have already been replaced by the specific $\cup$, $\supseteq$ and $\emptyset$ operators and constants as defined by the live variable analysis specification. The least fixpoint may then be computed by

Constraint	in_A	out_A	in_B	out_B	in_C	out_C
init	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
in_C	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{a_1, a_2\}$	$\emptyset$
out_B	$\emptyset$	$\emptyset$	$\emptyset$	$\{a_1, a_2\}$	$\{a_1, a_2\}$	$\emptyset$
in_B	$\emptyset$	$\emptyset$	$\{a_1, b_1\}$	$\{a_1, a_2\}$	$\{a_1, a_2\}$	$\emptyset$
out_A	$\emptyset$	$\{a_1, a_2, b_1\}$	$\{a_1, b_1\}$	$\{a_1, a_2\}$	$\{a_1, a_2\}$	$\emptyset$
in_A	$\{a_2\}$	$\{a_1, a_2, b_1\}$	$\{a_1, b_1\}$	$\{a_1, a_2\}$	$\{a_1, a_2\}$	$\emptyset$

after which all constraints are satisfied. The obtained variable assignment represented by the last table row is the result of the computation. As can be observed, at the end of block B the variable b_2 is not alive. Consequently the corresponding assignment operation may be dropped.

Limitations Although providing a vital utility for static program analysis in conventional compiler architectures favoring DFA based systems due to the close relation to common low-level compiler IRs and CFGs, data-flow analysis exhibit a few limitations – some of those hampering their applicability on the Insieme compiler IR. Those include:

- *Separation of Control-flow and Data-flow* – infrastructures following the DFA approach separate the control-flow, specified by the control-flow graph, from the data-flow covered by the derived constraints. Consequently, results of data-flow computations can not influence the control flow. For instance, in case a data flow analysis determines that a given branch of a conditional branch is never taken since the condition is a constant, the branch would still be considered as a potential path through the program since this result is not effecting the CFG. This lack of interaction leads to reduced accuracy in the obtained results which may only be compensated by an iterative approach including consecutive analysis and transformation passes or by extending the DFA framework itself.

- *Focused on intra-procedural analysis* – a control flow graph is, in general, only covering a single procedure/function body. Consequently "global" analysis and optimizations in conventional compilers are in general only referring to "covering a full procedure". *Inter-procedural* analysis are not covered by the basic framework, yet may be supported by extended variants merging the CFGs of multiple procedures into a single graph and add control-flow edges between call sites and the entry/exit points of procedures. Additional techniques introduce context information to distinguishing between different calls of the same procedures such that, to a certain extend, the effects of the various calls are not interfering with each other. Analyses empowered with those kind of capabilities are referred to as *context-sensitive* analyses.

Due to its functional roots, many primitives and extensions in the Insieme IR are modeled utilizing functions and most function bodies are more likely to be mere composition of nested function calls than sequences of simple instructions. As are most *object-oriented* codes. Consequently, powerful support for inter-procedural analysis is required for any framework to be practicable applied on our IR.

- *Dynamic Dispatch Problem* – related to the first two issues is this third issue. In languages where functions are first-class citizen, like in our IR, or dynamically called utilizing function pointers (C) or virtual function dispatching (C++), the function to be called at a given call site may not be statically fixed. Inter-procedural extensions of CFGs, however, may build on this assumption to identify procedures targeted by call expressions. Essentially, in such environments, functions are just another kind of data. Consequently, data-flow analyses can be utilized to determine which function may be called at which call site – yet those require a CFG. The separation of control- and data-flow representations as covered above is hampering the proper interaction and hence limiting the accuracy of analysis results and/or the applicability of the DFA approach in the first place.

- *Basic Blocks are formed by low-level instructions* – although not limiting in the general case, this property of DFAs is reducing its applicability on the Insieme IR. In a CFG basic blocks consist of sequences of three-address-code instructions, potentially restricted to SSA form, thereby providing a simple foundation for the automated extraction of the necessary transfer functions required for conducting analyses. However, our IR is based on high-level expressions likely consisting of multiple procedure calls. For applying conventional analyses those would have to be broken up into lists of simpler instructions, resulting in additional difficulties when mapping analysis results to the original IR node.

- *No parallel control-flow* – finally, DFAs are not prepared for parallel control flows. The underlying principle is designed to analyze the effects of sequentially processed instructions. Alternatives among control flow paths are always considered exclusive – hence one or the other, not both concurrently. However, to analyze an IR exhibiting parallel constructs, corresponding support is required.

  Nevertheless, ideas for integrating parallel control flows int DFAs have been presented in the past. For instance, pCFGs [108] add a second type of edge to the CFG indicating that those edges are always followed, thereby adding support for analyzing concurrent executions. Still, even with this extension, the parallel structure of the application has to be statically known during the construction of the pCFG. Yet, similar to the *dynamic dispatch problem*, the structure of the parallel application may be data-dependent. For instance, the entry point passed to a pthread_create call is always provided by a function pointer. Also, locks or other synchronization mechanisms may be referenced by pointers, resulting in an additional data-dependent element influencing the structure of the parallel program to be analyzed. While some work is presuming static knowledge regarding those relations [108, 40], more recent work focus on complex preparation steps to construct input graphs or similar structures for subsequent analysis steps [51, 115].

Several of those limitations and/or issues have been tackled by adding extra information to the CFG – e.g. edges to and from other procedures, parallel control flow edges or predicated edges to overcome the separation of control- and data-flow information. However, essentially those steps are customizing the structure providing the foundation for the (implicit) constraint generation leading to more flexible solutions. Yet, ultimately the creation of constraints could be completely detached from any (individual) structure to maximize flexibility. This is the basic idea of constraint-based analyses.

Constraint Based Analysis

The basic idea of constraint-based analysis (CBA) is to separate the generation of constraints from solving them, by making constraints explicit. The explicit representation of constraints in an intermediate step increases the flexibility of the constraint generation process. Different sources may be utilized. Furthermore, the format of constraints could (and is) enriched to improve the expressive power of the involved constraints. The resulting approach is best presented by an example.

Consider the following IR code fragment containing a data dependent control flow leading to a dynamic dispatched function call:

```
let int = int<4>;
int a = 5;
(int, int)→int f = (a<10)?+:−;
return f(2,a);
```

We chose INSPIRE as an example since the syntax and the semantic of the involved constructs has been introduced in the previous chapter. However, the presented approach can naturally be applied on other languages as well.

The challenge stated for an analysis is to determine the return value of this fragment. In a first step, every sub-term is labeled by a unique identifier:

```
let int = int<4>;
int [a]^1 = [5]^2;
(int, int)→int [f]^3 = [([[a]^4<[10]^5]^6)?[+]^7:[−]^8]^9;
return [[f]^10([2]^11,[a]^12)]^13;
```

Those labels are utilized to address the various sub-expressions. In the next step, for each property of interest, variables are assigned to the sub-expressions. In our example let

- I_x with property space $(2^{\mathbb{Z}}, \cup)$ be the integral value of the expression bearing the label x,
- B_x with property space $(2^{\mathbb{B}}, \cup)$ be the boolean value of the expression bearing the label x and
- F_x with property space $(2^{\mathcal{F}}, \cup)$ be the function value of the expression bearing the label x where $\mathcal{F}$ is the set of potential functions, e.g. $\mathcal{F} = \{+, -, *, /\}$

Based on those, constraints can be formulated. However, in addition to the shape

$$f(v_1, \ldots, v_n) \sqsubseteq v$$

for any analysis variables $v, v_1, \ldots, v_n$ utilized by DFAs, CBAs support the extended *conditional constraint* format

$$g(x_1, \ldots, x_n) \Rightarrow f(v_1, \ldots, v_m) \sqsubseteq v$$

where x_* and v_* are analysis variables, g is a n-ary *guard* predicate over the variables $x_1, \ldots, x_n$ and f is a function computing a *lower boundary* for the value to be assigned to v. Constraints of this shape state that $f(v_1, \ldots, v_m) \sqsubseteq v$ must hold in case the predicate $g(x_1, \ldots, x_n)$ is valid.

The constrains obtained from our input code include the following elements:

$$\begin{aligned}
I_2 &\subseteq I_1 \\
\{5\} &\subseteq I_2 \\
I_1 &\subseteq I_4 \\
\{10\} &\subseteq I_5 \\
\{2\} &\subseteq I_{11} \\
\{+\} \subseteq F_{10} \Rightarrow \{a + b \mid a \in I_{11} \wedge b \in I_{12}\} &\subseteq I_{13} \\
\{-\} \subseteq F_{10} \Rightarrow \{a - b \mid a \in I_{11} \wedge b \in I_{12}\} &\subseteq I_{13} \\
\{*\} \subseteq F_{10} \Rightarrow \{a * b \mid a \in I_{11} \wedge b \in I_{12}\} &\subseteq I_{13} \\
\{/\} \subseteq F_{10} \Rightarrow \{a/b \mid a \in I_{11} \wedge b \in I_{12}\} &\subseteq I_{13} \\
\exists (a, b) \in I_4 \times I_5 \,.\, a < b \Rightarrow \{true\} &\subseteq B_6 \\
\exists (a, b) \in I_4 \times I_5 \,.\, \neg(a < b) \Rightarrow \{false\} &\subseteq B_6 \\
\{+\} &\subseteq F_7 \\
\{-\} &\subseteq F_8 \\
true \in B_6 \Rightarrow F_7 &\subseteq F_9
\end{aligned}$$

$$\begin{aligned} \mathit{false} \in B_6 \Rightarrow F_8 &\subseteq F_9 \\ F_9 &\subseteq F_3 \\ F_3 &\subseteq F_{10} \end{aligned}$$

All of those can be obtained by converting each construct in the input code fragment into corresponding constraints. For instance, a *variable declaration* declaring a variable with label l utilizing an initial value labeled by k produces the constraint $X_k \sqsubseteq X_l$ for any type of analysis variable X and its associated relation operator $\sqsubseteq$. The constrain merely states that the values assigned to the declared variable must at least cover the values of its initializing expression. A subsequent variable reference labeled by an m results in a constraint $X_l \sqsubseteq X_m$, ensuring that expression m is considered to at least represent all the values the corresponding variable covers. Similar local rules can be fixed for all language constructs and operators.

Several cases, however, may result in more than a single constraint. In this example the boolean expression $a < b$ contributed two constraints and the call expression with the label 13 produced $|\mathcal{F}| = 4$ constraints. To be more concise, the call expression rule could actually be

$$\forall f \in \mathcal{F} \, . \, (f \in F_{10} \Rightarrow \{f(a,b) \mid a \in I_{11} \wedge b \in I_{12}\} \subseteq I_{13})$$

yet quantification is not included in the limited set of supported constraint formats. However, in any finite code, the number of functions to be considered is finite as well. And so is the set of potentially encountered functions $\mathcal{F}$. Therefore, the actually universally quantified rule can be substituted by all its instantiations as has been demonstrated above utilizing a finite over-approximation $\mathcal{F} = \{+, -, *, /\}$ of the present functions $\{+, -\}$. Note that in the general case also user defined functions need to be considered.

In our list of constrains we omitted elements including $B_2 \subseteq B_1$ or $I_3 \subseteq I_{10}$ which may equally be derived from the input code yet do not influence the following computation.

For our analysis we are interested in a variable assignment A representing a *least fixpoint* of the given constraints and in particular in the value assigned to the variable I_{13} in A.

Definition 4.6 (CBA constraint solver)**.** Let C be the set of constrains for which a least fixpoint shall be computed and C' the set of constraints obtained by replacing every constraint of the shape

$$f(v_1, \ldots, v_n) \sqsubseteq v$$

by the equivalent conditional constraint

$$\mathit{true} \Rightarrow f(v_1, \ldots, v_n) \sqsubseteq v$$

in C. Further, let V be the set of variables referenced by the constraints in C' and b a function mapping each variable $v \in V$ to the bottom element $\bot$ of its corresponding property space. Than the algorithm

```
// Step 1: init A with the property spaces' $\bot$ values
$A := \epsilon$
**for** $v \in V$ **do**
    $A := A[v \mapsto b(v)]$
**end for**

// Step 2: gradually fix unsatisfied constraints
**while** $\exists(c \Rightarrow t \sqsubseteq v) \in C' \,.\, A(c) \wedge (A(t) \not\sqsubseteq A[v])$ **do**
    $A := A[v \mapsto (A[v] \sqcup A(t))]$
**end while**
```

computes a least fixpoint assignment A for the constraints in C.

The algorithm essentially searches for unsatisfied constraints and updates the assignment accordingly. The predicates within the conditional constraints may thereby be utilized for selecting constraints to be obeyed. As for the DFA, limitations on the set of functions utilized within constraints have to be imposed to ensure the existence of a least fixpoint solution. Among the most severe is the restriction to monotone functions – for both, the guard predicate and the function computing a lower boundary of the constraint variable. Another is, as for DFAs, the *Ascending Chain Condition* ensuring the algorithm to terminate within a finite number of steps [68].

The following table covers the least fixpoint assignment obtained by this algorithm when being applied on the constraints listed above.

I_1	I_2	I_4	I_5	I_{11}	I_{13}	B_6	F_3	F_7	F_8	F_{10}
$\{5\}$	$\{5\}$	$\{5\}$	$\{10\}$	$\{2\}$	$\{7\}$	$\{true\}$	$\{+\}$	$\{+\}$	$\{-\}$	$\{+\}$

The analysis could successfully deduce the resulting value being $I_{13} = \{7\}$.

Note that the presented CBA constraint solver algorithm is handling variables of various property spaces simultaneously while the DFA algorithm is limited to a single property space. Furthermore, notice the interaction of control flow and data flow in the CBA constraints. Results of data flow analysis are utilized to activate data-flow dependent constraints, thereby effectively constraining the number of considered paths. A standard DFA analysis would have determined the value of the $a < 10$ expression, yet the variable f would still get the uncertain value $\{+, -\}$ assigned after decomposing the conditional operator into three basic blocks including the evaluation of the condition, the two branches and at least one temporary variable.

Also, the example demonstrated that the same mechanism utilized to constraint control-flow paths also provides a solution for solving the dynamic dispatch problem – by treating functions like data. This support is not only limited to primitive functions as included in the given example. Functions created by arbitrary lambda or closure expressions may be treated as well, including the necessary parameter and return value passing.

Due to all its benefits, the constraint-based approach has been utilized as the foundation of the analysis framework established for the development of flow- and context-sensitive analyses processing programs encoded using the high-level Insieme IR.

4.4.2 Overview on the Insieme CBA Framework

The framework established for analyzing INSPIRE codes is based on the CBA framework presented above, yet several details have been extended and customized to gain increased flexibility to suite the specific requirements as well as increased accuracy.

Overview on solver modifications

Among the most significant modifications is the advancement of a conventional constraint solver approach, depending on a complete list of constraints as an input, to a *lazy* approach, capable of incorporating constrains on demand. On the one hand, the development of a *lazy-solver* was motivated by the observation that only a tiny portion of the set of constraints to be extractable from a given code fragment is required to obtain desired analysis results. This is due to the fact that variables may describe irrelevant properties like the function value of a integer constant – e.g. I_3 in the example of the previous sub-section – or results of unreachable program states cut off from the relevant part of the system of constraints by unsatisfied predicates. In particular when introducing call context information the vast majority of call-stack approximations are unreachable and all its associated variables are hence irrelevant for the analysis.

On the other hand a lazy solver enables analysis to react on temporary results. In particular intermediate results may be utilized to form the foundation for the creation of new constraints influencing the enclosing computation of a least fixpoint solution – hence the result of an analysis. As will be demonstrated, this feature is utilized to conduct analysis on the parallel structure of codes where the structure description itself is a result obtained as part of the same analysis.

Furthermore, the solver algorithm has been modified to eliminate the strict requirement of monotone *guardian* predicates and *lower boundary* functions to obtain more accurate analysis results. In general, monotone predicates and functions are sufficient to accurately model the behavior of many language constructs and lead to a fast conversion of the solver algorithm. Therefore they are utilized for the vast majority of constraints. However, in some cases, restrictive assumptions on values can lead to more accurate results which, however, may be invalidated over the course of the conversion process. In such a case the assumption has to be eliminated, previously obtained and related results "forgotten" and revised without considering this assumption. The only restriction, however, is that such reset-events are not repeatedly triggering each other – a property of the constraints in other contexts referred to as *stratification*. Essentially the original monotonicity condition on the individual elements of constraints has been lifted to a global monotonicity condition addressing the full set of constraints. Support for such a "reconsider" mechanism has been integrated into our constraint solver to further increase the accuracy of analysis built on top of it.

Details on the supported constrain format, our lazy constraint solver algorithm and its properties are covered in more detail in Section 4.4.3.

Analysis Variables

Unlike in the example demonstrated in the CBA introduction, the Insieme CBA framework is utilizing multiple structures to associated variables to. Those include:

- *Labeled Expressions* – to represent the value of an expression as covered in the CBA introduction, however, extended by thread and call context information

- *Program Points* – to represent the state of some environment object, e.g. the value of mutable memory location, before, during or after the evaluation of a labeled expression

- *Whole-Program* or *Global* information – analysis results describing the complete program (fragment) to be analyzed, e.g. the set of synchronization points or the parallel structure of the code fragment

- *Program State Graph Nodes* – the nodes of the graph describing the parallel structure of an application; those are, for instance, utilized to represent the state of channels which can not be associated to individual program points

In the following we will provide definitions for those structures. However, future extensions of the framework may even extend this list by additional structures.

Labeled Expressions

In the introduction to CBA based systems all the sub-terms of a given code fragment have been labeled to reference them and to associated analysis variables to them. The same concept is utilized in the Insieme CBA framework, yet instead of using natural numbers, labels are realized by *node instances*, a variant of node addresses introduced in Section 3.10.2, combined with context information capturing the thread and function-call context.

We start by provide a definition for the *node instances.* A node instance is, similar to a *node address*, referencing a substructure within an IR tree by the path from a given root node to the targeted structure. However, additionally, whenever passing a loop along the path the iteration of the loop is specified. This way instances of the some IR sub-structure being processed in different loop iterations can be addressed.

However, since the number of loop iterations is in general statically unknown, an abstract representation is required.

Definition 4.7 (abstract loop iterator)**.** An abstract loop iterator is given by a total ordered set defined by the pair

$$(I, \leq)$$

where I is an arbitrary, finite set forming the domain of the iterator and $\leq: I \times I$ is total order on the iterator values. Further, let $\bot$ be the minimal element such that $\forall x \in I \,.\, \bot \leq x$ and $\top$ be the maximal element such that $\forall x \in I \,.\, x \leq \bot$.

A simple example of an abstract loop iterator is given by the pair

$$(\{*\}, \{(*, *)\})$$

which only utilizes a single token $*$ to summarize all loop iterations. Accordingly we have $\bot = \top = *$. A more sophisticated representation is given by

$$(\{0, 1, 2, \ldots, n, *, -n, \ldots, -2, -1\}, \leq)$$

where n is some fixed integer and $\leq$ is defined according to the order the elements in the set have been listed. The value $\bot = 0$ is addressing the first iteration, the value 1 the second, -2 the second last and $-1 = \top$ the last iteration. The $*$ token represents any remaining iteration. This loop iterator abstraction enables analysis, to some extend, to distinguish individual loop iterations within a targeted code fragment.

The actual abstract loop iterator to be chosen may be customized to fit the requirements of the analysis to be conducted, e.g. by using some advanced symbolic representations as an abstract iterator. However, for simplicity, in the remainder of this chapter we will assume loop iterators to be modeled utilizing something similar to the second approach outlined above with e.g. $n = 2$.

Based on those we can define node instances as follows:

Definition 4.8 (node instance)**.** A *node instance* is a term of the grammar

$$i ::= r \mid i.x \mid i.x[j]$$

where $r \in \mathbb{E} \cup \mathbb{S}$ is an IR expression or statement, $x \in \mathbb{N}_0$ is and index and $j \in I$ is an abstract loop iterator of type I. Let the set of all node instances be denoted by $\mathcal{I}'$. Also, for all $n \in \mathbb{E} \cup \mathbb{S}$ and $x \in \mathbb{N}_0$ let $n[x]$ denote the child node of n at index x where indices start at 0. If n has less than $x + 1$ child nodes $n[x]$ is $undefined$. Then the function $\text{node} : \mathcal{I}' \to (\mathbb{E} \cup \mathbb{S})$ determines the IR sub-structure referenced by a node instance $i \in \mathcal{I}'$ and is defined by

$$\text{node}(i) = \begin{cases} r & \text{if } i = r \in \mathbb{E} \cup \mathbb{S} \\ \text{node}(i')[x] & \text{if } i = i'.x \text{ or } i = i'.x[j] \end{cases}$$

Furthermore, a node instance i is valid iff $\text{node}(i)$ is defined and

- $i = r \in \mathbb{E} \cup \mathbb{S}$ or
- $i = i'.x$ and i' is valid and $\text{node}(i')[x]$ is not the body of a while or for loop construct or
- $i = i'.x[j]$ and i' is valid and $\text{node}(i')[x]$ is the body of a while or for loop construct

The set of all valid node instances is denoted by $\mathcal{I}$.

Node instances provide means to address sub-expressions and statements contained in a given IR code fragment, similar to the labels introduced in the example covered in the CBA introduction section. By labeling each sub-structure using the path it can be reached by from a common root node, all labels are automatically distinct. Also, common sub-expressions can be distinguished and due to the integration of abstract loop iterators identical sub-expressions processed within different loop iterations may be distinguished.

Example 4.7 (node instances)**.** Consider the compound statement c given by

```
{
   a[0] = 0;
   for(int<4> i = 0 .. 10 : 1) {
      a[i] := a[i]+10;
   }
}
```

The node instance $c \in \mathcal{I}$ references the full compound statement, $c.0 \in \mathcal{I}$ the first statement $a[0] = 0$, $c.1$ the for loop and $c.1.4[1] \in \mathcal{I}$ the compound statement of the 2nd iteration of the for loop since the 5th child of the for loop is its loop body. Also, $c.1.4[-2].0 \in \mathcal{I}$ references the assignment $a[i] := a[i] + 10$ of the second last iteration of the loop. To the contrast, $c.2 \in \mathcal{I}'$ is not a valid node instance since c does not have a child node with the index 2.

A side effect of our IR structure is that statically bound functions are immediately present at the call site. Consequently, since *node addresses* and *node instances* are describing the path from a common root node to the targeted sub-structures, expressions within bodies of functions directly bound at some call side are implicitly associated with a call context.

Example 4.8 (implicit call context)**.** Consider the for loop f given by

```
for(int<4> i=0 .. 10 : 1) {
   a[i] := (int<4> a)→int<4> {
      return a + 10;
   }(a[i]);
}
```

The node instance $f.4[0].0.2 \in \mathcal{I}$ references the function call on the right-hand side of the assignment in the first loop iteration, $f.4[0].0.2.0.1 \in \mathcal{I}$ the body of the function and $f.4[0].0.2.0.1.0.0 \in \mathcal{I}$ the return value of the function invocation in the left hand side of the assignment operation processed by the first iteration of the loop. The node instance $f.4[1].0.2.0.1.0.0 \in \mathcal{I}$ references the same element yet for the second iteration of the loop. The actual indices utilized to address sub-structures of the involved constructs are not important for this example. Important is that the address includes the full call context of the directly invoked function.

However, not all function calls are direct, statically bound function calls since functions may be recursive – on those cases *recursive variables* are utilized – or forwarded as first-class citizens to the call site. For those cases explicit support for recording call contexts is required.

Example 4.9 (required call context)**.** Consider the compound statement c given by

```
{
   auto f = (int<4> a)→int<4> {
      return a + 10;
   };
   for(int<4> i=0 .. 10 : 1) {
      a[i] := f(a[i]);
   }
}
```

Here the function f is not directly called. The address of the return value of the function is given by $c.0.2.1.0.0$ addresses the return value where $c.0$ is the declaration statement and $c.0.2$ the lambda expression. The invocation of the function in the first loop iteration, for instance, is referenced by $c.1.4[0].0.2$. In this case the context is not implicit and has to be modeled explicitly.

To record the context of non-statically bound function calls we utilize an abstract representation of the corresponding call-stack which is simply recording up to the last n dynamic function call sites passed to reach the addressed structure.

Definition 4.9 (call context)**.** Let $n \in \mathbb{N}$ be the maximum number of passed call sites to be used to distinguish contexts of function calls. Let $\mathcal{I}_{dyn\_call}$ be the set of node instances referencing call expressions targeting non-statically bound functions. Then the set of *call contexts* $\mathcal{C}$ is given by

$$\bigcup_{0 \leq i \leq n} (\mathcal{I}_{dyn\_call})^i$$

Hence, a context $c = [c_1, \ldots, c_m] \in \mathcal{C}$ is a sequence of up to n node instances also referred to as a *call string*. Further, we define the function push : $(\mathcal{C} \times \mathcal{I}) \to \mathcal{C}$ by

$$\text{push}(c, i) = \begin{cases} [] & \text{if } n = 0 \\ [c_2, \ldots, c_n, i] & \text{if } c = [c_1, \ldots, c_n] \\ [c_1, \ldots, c_m, i] & \text{if } c = [c_1, \ldots, c_m] \text{ and } m < n \end{cases}$$

The context $c_{in} = \text{push}(c_{call}, i)$ is the call context reached when a call expression referenced by $i \in \mathcal{I}$ is invoking in context c_{call} a non-statically bound function. The body of the targeted function is then evaluated utilizing the call context c_{in}.

The parameter n determines the maximum length of the call stack to be recorded. If set to 0 every expression will be evaluated in the empty call context [] – hence no explicit call context is considered, only the implicit context captured by the addressing utilizing *node instances*. Larger values will potentially improve the accuracy of analysis results, yet also increases the number of variables and hence the run-time complexity of the analysis.

Example 4.10 (call strings)**.** Let $c_1, c_2 \in \mathcal{I}$ be two node instances referencing call expressions targeting non-statically bound functions. Further, let the maximum length of call contexts n be 2. When analyzing the enclosing code fragment the initial call context for any expression is the empty context $[] \in \mathcal{C}$. When the control flow passes c_1, the body of the targeted function will be analyzed utilizing the call context $[c_1]$. If, while doing so,

c_2 is passed, the targeted nested function body will be evaluated using the context $[c_1, c_2]$. However, if in a third nesting level c_2 would be reached a second time, e.g. at a recursive function call site, the new nested context is $[c_2, c_2] \in \mathcal{C}$ instead of the more accurate $[c_1, c_2, c_2] \notin \mathcal{C}$. This is due to the restriction on the maximum length of the utilized call strings since unlimited call strings could lead to an infinite number of variables and constraints.

If c_1 and c_2 are the only dynamic call sites in the analyzed code fragment, the only feasible call contexts are $[]$, $[c_1]$, $[c_2]$, $[c_1, c_1]$, $[c_1, c_2]$, $[c_2, c_1]$ and $[c_2, c_2]$ – hence a finite number of contexts. Any function call stack will be mapped to one of those 7 abstract stack representations. In case of collisions, analysis results may become less accurate, yet soundness is not affected. Still, the introduction of call contexts is significantly increasing the accuracy of program analysis. The effects of collisions in the associated context may further be reduced by increasing the maximum string length if required.

Besides the execution of the same expression in the body of a function in different call contexts it is also required to distinguish instances within different threads. The Insieme IR features explicit thread support and hence concurrent control flows need to be covered. We therefore introduce, analogous to call contexts, *thread contexts*.

To describe the context of a function we have been aggregating strings of node instances referencing call expressions. To address threads we utilize finite strings of thread-spawn points – hence invocations of the *parallel* construct – to identify threads.

Definition 4.10 (thread context)**.** Let $m \in N$ be the maximum number of spawn points to be considered to distinguish thread contexts. Let $\mathcal{I}_{spawn}$ be the set of node instances referencing call expressions potentially creating a thread group, hence potential calls to the *parallel* primitive. Then the set of *thread contexts* $\mathcal{T}$ is given by

$$\bigcup_{0 \leq i \leq m} (\mathcal{I}_{spawn} \times \mathcal{C} \times \mathbb{N})^i$$

Hence, a thread context $t = [(p_1, c_1, i_1), \ldots, (p_k, c_k, i_k)] \in \mathcal{T}$ is a sequence of up to m spawn expressions, their thread-local call contexts and the IDs of the processing threads within their groups. As for the call context, we further define the function push : $(\mathcal{T} \times \mathcal{I} \times \mathcal{C} \times \mathbb{N}) \to \mathcal{T}$ by

$$\text{push}(t, s, c, i) = \begin{cases} [] & \text{if } m = 0 \\ [t_2, \ldots, t_m, (s, c, i)] & \text{if } t = [t_1, \ldots, t_m] \\ [t_1, \ldots, t_k, (s, c, i)] & \text{if } t = [t_1, \ldots, t_k] \text{ and } k < m \end{cases}$$

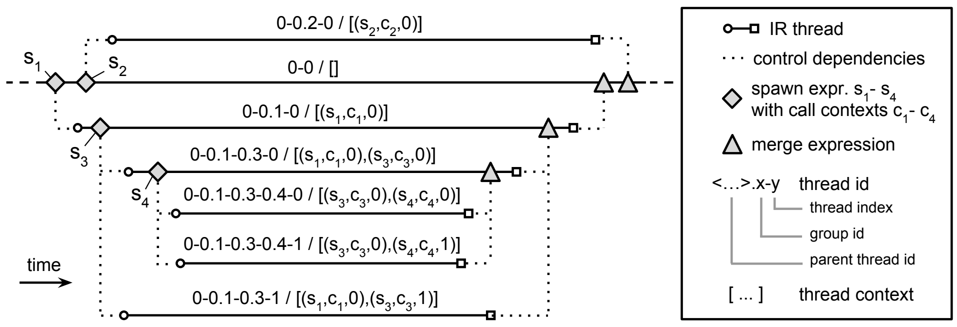

0-0.2-0 / [(s2,c2,0)]
0-0 / []
0-0.1-0 / [(s1,c1,0)]
0-0.1-0.3-0 / [(s1,c1,0),(s3,c3,0)]
0-0.1-0.3-0.4-0 / [(s3,c3,0),(s4,c4,0)]
0-0.1-0.3-0.4-1 / [(s3,c3,0),(s4,c4,1)]
0-0.1-0.3-1 / [(s1,c1,0),(s3,c3,1)]
s1
s2
s3
s4
time
IR thread
control dependencies
spawn expr. s1- s4 with call contexts c1- c4
merge expression
<...>.x-y thread id
thread index
group id
parent thread id
[...] thread context

Figure 4.1: Comparison of thread IDs and thread contexts.

The thread context $t_{child} = \text{push}(t_{parent}, s, c, 2)$ is the thread context of the thread with index 2 created by the thread-spawning expression s processed in call context c and thread context t_{parent}. The body of the spawned thread is then evaluated utilizing the thread context t_{child}.

Example 4.11 (thread contexts)**.** Figure 4.1 visualizes nested groups of threads. Each thread is labeled by its hierarchical ID, as it has been introduced in Section 3.3.2, and the thread context utilized in the Insieme CBA framework for addressing and distinguishing those. The maximum length of thread contexts is limited to $m = 2$. In the example, the statement s_1 processed by thread 0-0 in the call context c_1 spawns thread 0-0.1-0. Since the thread context of the main 0-0 is the empty sequence $[] \in \mathcal{T}$, the thread context addressing the spawned thread is $\text{push}([], s_1, c_1, 0) = [(s_1, c_1, 0)] \in \mathcal{T}$. Similarly, the thread context thread 0-0.2-0 is mapped to is $\text{push}([], s_2, c_2, 0) = [(s_2, c_2, 0)] \in \mathcal{T}$ and the nested threads 0-0.1-0.3-0 and 0-0.1-0.3-1 spawned by thread 0-0.1-0 are addressed by $[(s_1, c_1, 0), (s_3, c_3, 0)]$ and $[(s_1, c_1, 0), (s_3, c_3, 1)]$ in the Insieme CBA framework. When reaching nesting levels greater than $m = 2$ the length of the thread context sequences reaches its limits. Consequently, the threads 0-0.1-0.3-0.4-0 and 0-0.1-0.3-0.4-1 are associated to the thread contexts $[(s_3, c_3, 0), (s_4, c_4, 0)]$ and $[(s_3, c_3, 0), (s_4, c_4, 1)]$ referencing only the two innermost passed spawning points.

As for the call contexts, the set of possible thread contexts is finite since the set of spawn points in any program is finite and thus the set of call contexts too. The full list, however would encompass an astronomically high number of values for every practical example. Fortunately, in meaningful code fragment only a very small number of contexts are reachable and hence need to be considered due to the lazy-constraint solver algorithm covered in the Section 4.4.3.

Note that thread contexts are abstract, finite approximations of the *thread addresses* utilized for defining the semantic of IR constructs in Section 3.7.1 on page 94.

Finally we can define the format of a label utilized by the Insieme CBA framework to address expressions within a code fragment.

Definition 4.11 (node labels)**.** An expression or statement of a code fragment to be analyzed is addressed by *node label l* defined by a tuple

$$l = (i, c, t)$$

where $i \in \mathcal{I}$ is a node instance, $c \in \mathcal{C}$ a call context in a processing thread and $t \in \mathcal{T}$ a thread context describing the processing thread. The set of all node labels is denoted by $\mathcal{L}$.

Program Points

Another object to be utilized to attach variables to are *program points*. Variables attached to *labeled expressions* are utilized to characterize the value computed by this expression within its context. However, they can not be utilized to describe the state of e.g. a mutable value at a given program point. Fortunately, program points can be directly derived from node labels.

Definition 4.12 (program points)**.** A *program point p* is a pair

$$p = (l, s) \in \mathcal{L} \times \{\text{pre}, \text{in}, \text{post}\}$$

where $l = (i, c, t) \in \mathcal{L}$ is a node label referencing an expression ($i \in \mathcal{I}$) and its call ($c \in \mathcal{C}$) and thread ($t \in \mathcal{T}$) context, and $s \in \{\text{pre}, \text{in}, \text{post}\}$ determines whether the program point before (pre), during (in) or after (post) the processing of the referenced expression is to be addressed. The set of all program points is denoted by $\mathcal{P}$.

Example 4.12 (program points)**.** Let $i \in \mathcal{I}$ be some node instance, $c \in \mathcal{C}$ a call context and $t \in \mathcal{T}$ a thread context. The program point $((i, c, t), \text{pre})$ references the state before evaluating the expression i in context c of thread t. The evaluation of the expression may lead to the post state which is addressed by the program point $((i, c, t), \text{post})$. If $i' \in \mathcal{I}$ is the expression to be evaluated after expression i than $((i, c, t), \text{post})$ is the predecessor of $((i', c, t), \text{pre})$. For some operations an intermediate step between the pre and post state is required. For instance, to reference the point in which arguments of a function have been evaluated yet the call has not yet been conducted. In those cases in states may be utilized to obtain the processing order (l, pre), (l, in) and (l, post) for some label $l \in \mathcal{I}$.

Global Information

The third and simplest entity to attach values to is the full program. For this entity we do not have to introduce any constructs. Analysis variables characterizing the full program are individual variables not to be bound to any structure by any subscript.

Program State Graph Nodes

The last type of variables utilized in the Insieme CBA framework are variables attached to nodes of the *program state graph*.

A *program state graph* describes the regions of a program which may be executed concurrently. It is derived from an *execution net*, which is a labeled extension of a *Petri net* [81] and constructed based on a set of *synchronization points* to be derived from the analyzed code segment. In the following we provide definitions for those entities, along with an example.

Consider the following code fragment including three threads, a mutable memory location a and a channel c:

```
auto a = var(0);
auto c = channel.create(int<4>,2);
auto t1 = parallel(job[1,1]()=> {
  a = 1;
  channel.send(c,1);
});
auto t2 = parallel(job[1,1]()=> {
  channel.recv(c);
  a = 2;
});
a = 3;
merge(t1);
merge(t2);
```

The first thread, the “main” thread A, creates a variable a and a channel c, spawns two threads, updates a and merges the two previously spawned thread groups. Each group consisting of a single thread ($t1 = B$ and $t2 = C$ respectively). Thread B updates a and sends a message through channel c while the C waits for a message before also updating a. In this code fragment we could be interested in whether there are *race conditions*, *dead locks*, or in the data that is transferred through the channel. The first step towards an infrastructure enabling all those analyses is to identify all synchronization points within the code fragment.

Definition 4.13 (synchronization points)**.** *Synchronization points* – also *sync points* – are program points (see Definition 4.12) referring to one of the following events:

- the begin / end of a thread
- a call to the parallel or merge operators
- a call to the send or recv channel operators
- a call to the redistribute operator
- the begin / end of the analyzed code fragment

The set of all sync points is denoted by $\mathcal{S} \subset \mathcal{P}$.

Example 4.13 (sync points). For simplicity, to avoid the cumbersome syntax of node instances and node labels, let the following numerical labels be assigned to the constructs of the example introduced above:

```
[
  auto a = var(0);
  auto c = channel.create(int<4>,2);
  auto t1 = [parallel(job[1,1]()⇒ [{
    a = 1;
    [channel.send(c,1)]^6;
  }]^2)]^4;
  auto t2 = [parallel(job[1,1]()⇒ [{
    [channel.recv(c)]^7;
    a = 2;
  }]^3)]^5;
  a = 3;
  [merge(t1)]^8;
  [merge(t2)]^9;
]^1
```

Further, let those labels include the call and thread contexts according to the composition of the involved language constructs. Than we have the following sync points:

- $(2, \text{pre})$ and $(2, \text{post})$ – begin and end of thread 1
- $(3, \text{pre})$ and $(3, \text{post})$ – begin and end of thread 2
- $(4, \text{in})$, $(5, \text{in})$, $(8, \text{in})$ and $(9, \text{in})$ – calls to parallel and merge
- $(6, \text{in})$ and $(7, \text{in})$ – calls to send/recv operations
- $(1, \text{pre})$ and $(1, \text{post})$ – begin and end of code fragment

Note that the labels $1, 4, 5, 8$ and 9 exhibit the same thread context, as do the labels 2 and 6 as well as 3 and 7. However, e.g. 1 and 2 do not.

Next the obtained sync points are combined into *thread regions.*

Definition 4.14 (thread region)**.** A thread region $(a, b) \in \mathcal{S}^2$ is a pair of sync points $a \in \mathcal{S}$ and $b \in \mathcal{S}$ such that the program point b can be reached from program point a without passing any other sync point. The set of thread regions is denoted by $\mathcal{R}$.

Since the begins and ends of a threads are sync points and are the only places where thread contexts may change, only sync points referencing labels including the same thread context may constitute thread regions. Hence, only sequential control flow between sync points needs to be considered.

Example 4.14 (thread regions)**.** For our running example we obtain the following thread regions for thread A

$$\begin{aligned} A_1 &= ((1, \mathrm{pre}), (4, \mathrm{in})) \\ A_2 &= ((4, \mathrm{in}), (5, \mathrm{in})) \\ A_3 &= ((5, \mathrm{in}), (8, \mathrm{in})) \\ A_4 &= ((8, \mathrm{in}), (9, \mathrm{in})) \\ A_5 &= ((9, \mathrm{in}), (1, \mathrm{post})) \end{aligned}$$

for thread B

$$\begin{aligned} B_1 &= ((2, \mathrm{pre}), (6, \mathrm{in})) \\ B_2 &= ((6, \mathrm{in}), (2, \mathrm{post})) \end{aligned}$$

and for thread C we have

$$\begin{aligned} C_1 &= ((3, \mathrm{pre}), (7, \mathrm{in})) \\ C_2 &= ((7, \mathrm{in}), (3, \mathrm{post})) \end{aligned}$$

Note that within a thread region no synchronization event can happen. Also no channel state could be modified.

In the following step the coordination and relation between thread regions has to be described. The resulting description is a *execution net* which is a variant of a *Petri net*.

Petri nets are bipartite graphs well established for modeling concurrent and distributed systems [81]. For completeness we provide a formal definition of such a net.

Definition 4.15 (Petri net graph with place capacities)**.** A *Petri net graph with place capacities* is given by a tuple

$$(P, T, E, \mathrm{cap})$$

where P is an arbitrary set of *places*, T a set of *transitions* such that $P \cap T = \emptyset$, $E \subseteq (P \times T) \cup (T \times P)$ a set of directed edges connecting places and transitions and $\mathrm{cap} : P \to \mathbb{N}_{>0}$ a function assigning each place a capacity.

The general idea is to model the execution of a code fragment utilizing a Petri net where thread regions are the places and sync points the transitions. Each thread region has a capacity limited to 1. Furthermore, additional places modeling channel buffers are introduced for individual channel instances such that their capacity corresponds to the static buffer sizes of the associated channels. Furthermore, occasionally, auxiliary places and transitions have to be introduced to model uncertainty in cases where more accurate information is not available. Auxiliary places also have a fixed capacity of 1.

Definition 4.16 (execution net)**.** Let $C \subset \mathcal{P}$ be a set of program points creating channels, hence calls to the *channel.create* operator, and A_p and A_t be two arbitrary set such that $A_p \cap \mathcal{R} = A_p \cap C = \emptyset$ and $A_t \cap \mathcal{S} = \emptyset$, where $\mathcal{R}$ is the set of thread regions and $\mathcal{S}$ the set of sync points. An *execution net* is a *Petri net graph with place capacities*

$$(P, T, E, \text{cap})$$

where places

$$P \subset \mathcal{R} \cup C \cup A_p$$

are program regions ($\mathcal{R}$), channels identified by their creation points (C) or auxiliary places (A_p) and transitions

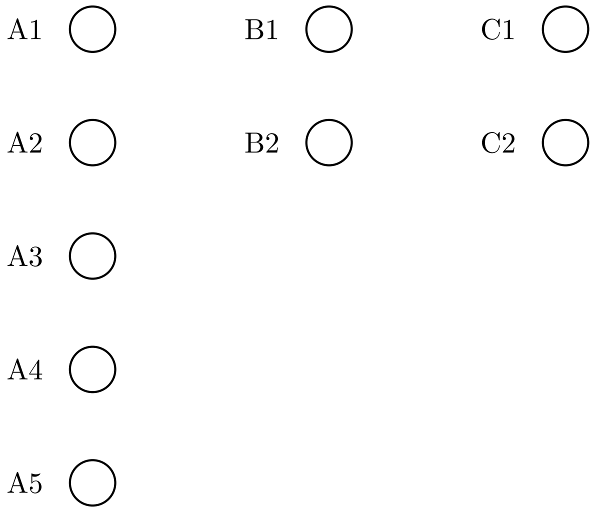
$$T \subset \mathcal{S} \cup A_t$$

are either sync points ($\mathcal{S}$) or auxiliary transitions (A_t). The function cap is defined by

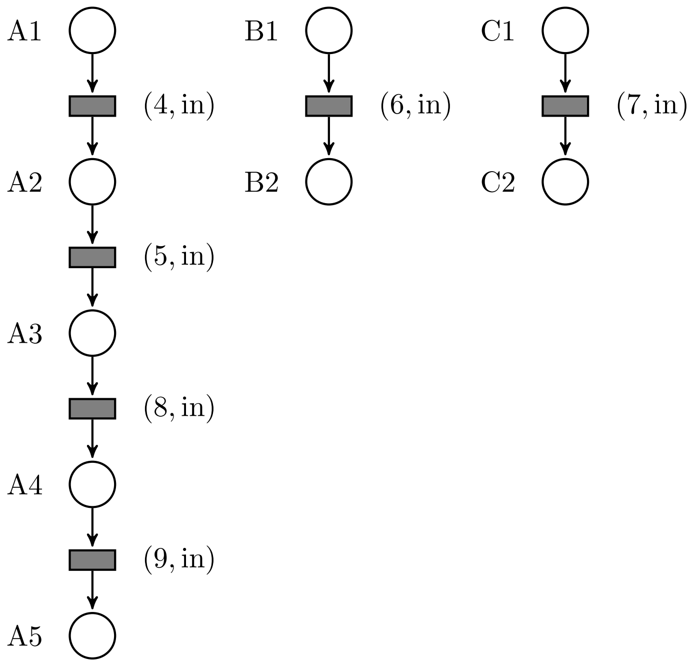
$$\text{cap}(p) = \begin{cases} 1 & \text{if } p \in \mathcal{R} \\ \text{capacity}(p) & \text{if } p \in C \end{cases}$$

where capacity : $C \to \mathbb{N}_{>0}$ determines the capacity of a channel which is statically defined by its type.

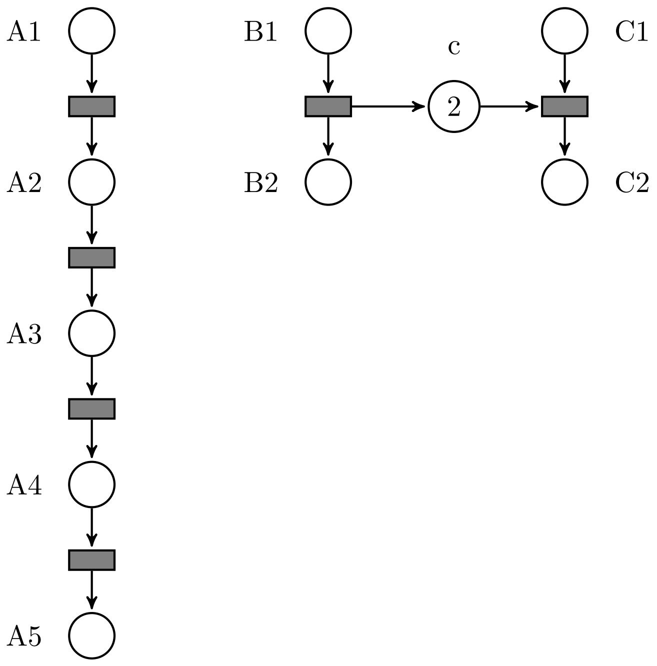
Example 4.15 (execution net)**.** The execution net of our running example can be constructed as follows. We start with the set of thread regions already obtained from the targeted code fragment. Hence we have obtain



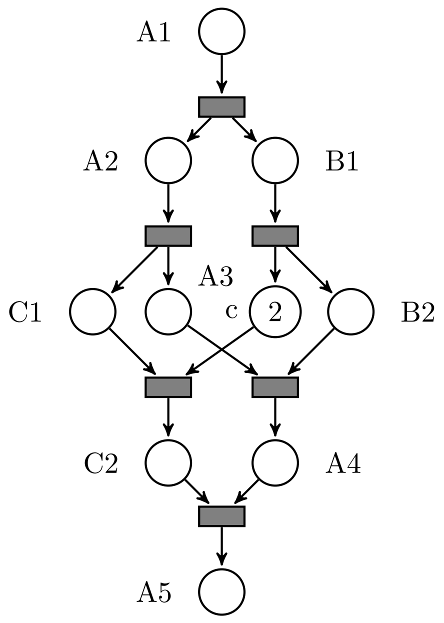
Next, for each sync point in the code fragment that is not the begin or end of a thread or the analyzed code fragment we add a transition to the graph connecting adjacent thread regions to obtain



In a next step we add the place representing the channel and connect it to send and receive operations. We obtain



where we dropped the labeling of transitions for clarity. Unlike the other places, the capacity of the channel state may be bigger than 1. In the given case its capacity is 2, which corresponds to its buffer size. In a final step sync points spawning or merging threads are connected to the begin or end of the corresponding threads. In our example this results in



describing the dependencies between the involved thread regions.

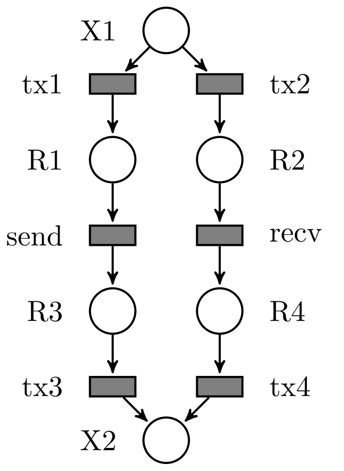
Example 4.16 (execution graph with uncertainty)**.** In the previous example no auxiliary places and transitions have been necessary. However, to provide a basic idea when those are necessary, consider the following labeled code fragment:

```
parallel(job[1,1]()⇒ [{
   if (<something>) {
      [channel.send(c,2)]²;
   } else {
      [channel.recv(c)]³;
   }
}]¹);
```

The inner thread constitutes 4 thread regions

$$\begin{aligned}
R1 &= ((1, \text{pre}), (2, \text{in})) \\
R2 &= ((1, \text{pre}), (3, \text{in})) \\
R3 &= ((2, \text{in}), (1, \text{post})) \\
R4 &= ((3, \text{in}), (1, \text{post}))
\end{aligned}$$

since the condition expression can not be determined statically. Besides others, it can not be determined statically whether $R1$ or $R2$ is entered when entering a thread. In those cases in-deterministic choices are added to the execution net utilizing auxiliary places and transitions similar to



Here $X1$,$X2$,$tx1$,...,$tx4$ are auxiliary places and transitions. $X1$ becomes the entry point of the thread to be connected to the parallel call of its parent node and $X2$ the end point to be connected to a merge call. Other cases caused by unknown information are covered using similar solutions.

Naturally, the introduction of auxiliary nodes may lead to less accurate results. However, any program analysis necessarily has to be restricted to approximations of the actual behavior encountered during the execution of the analyzed code. The important restriction is that analyses, depending on their objectives, produce over- or under-approximations of the actual results. Hence, e.g. value analyses may result in to much possibilities, yet they must never exclude values which may actually occur (over-approximation). The approach of dealing with uncertainty in analyses outlined in the example above is over-approximating the actual observable control flows since in an actual execution only one of the two alternatives may be followed.

In a final step we obtain the *program state graph* from the *execution graph* by computing the *reachability graph* according to the rules determined by the underlying Petri net graph starting from an initial *marking* covering only the entry region of the analyzed code fragment.

Definition 4.17 (markings)**.** Let $g = (P, T, E, cap)$ be a Petri net graph. A valid marking m is a mapping $P \to \mathbb{N}$ such that

$$\forall p \in P \,.\, m(p) \leq \mathrm{cap}(p)$$

Let M be the set of all valid markings of g, $\mathrm{p} : T \to 2^P$ the function mapping each transition in g to its predecessors and $\mathrm{s} : T \to 2^P$ the function mapping each transition in g to its successors. Further, let the step relation

$S \subset M \times T \times M$ be defined by

$$\begin{aligned} S =& \{(m,t,n) \in M \times T \times N \mid \\ & \quad (\forall x \in \mathrm{p}(t) \ . \ m(x) > 0) \wedge \\ & \quad (\forall x \in \mathrm{s}(t) \ . \ m(x) < cap(x) \vee x \in \mathrm{p}(t)) \wedge \\ & \quad (\forall x \in P \ . \ n(x) = m(x) - \kappa_{p(t)}(x) + \kappa_{s(t)}(x)) \\ & \} \end{aligned}$$

where $\kappa_S(x)$ is defined by

$$\kappa_S(x) = \begin{cases} 0 & \text{if } x \notin S \\ 1 & \text{if } x \in S \end{cases}$$

Then the *transition relation* $\rightarrow_g \in M \times M$ is given by

$$\{(m,n) \in M \times M \mid \exists t \in T \ . \ (m,t,n) \in S\}$$

and connecting all markings m with possible markings n that can be reached by *firing* a single transition.

In this section we only provided a brief introduction on Petri nets and the associated semantic definitions. Interested readers may be referred to the literature [81].

Definition 4.18 (program state graph)**.** A *program state graph* of a given *execution net* g is its *transition relation* $\rightarrow_g$ restricted to reachable markings seeded by a marking assigning 1 to the thread region starting at the beginning of the analyzed code fragment.

The *program state graph* therefore describes the *state space* of the *execution net.*

Example 4.17 (program state graph)**.** To complete our running example we obtain the following *program state graph* from the previously obtained *execution graph*

$$\begin{tikzcd}
& \boxed{\{A_1\}, c = 0} \arrow[d] & \\
& \boxed{\{A_2, B_1\}, c = 0} \arrow[dl] \arrow[dr] & \\
\boxed{\{A_3, B_1, C_1\}, c = 0} \arrow[dr] & & \boxed{\{A_2, B_2\}, c = 1} \arrow[dl] \\
& \boxed{\{A_3, B_2, C_1\}, c = 1} \arrow[dl] \arrow[dr] & \\
\boxed{\{A_4, B_2, C_1\}, c = 1} \arrow[dr] & & \boxed{\{A_3, B_2, C_2\}, c = 0} \arrow[dl] \\
& \boxed{\{A_4, C_2\}, c = 0} \arrow[d] & \\
& \boxed{\{A_5\}, c = 0} &
\end{tikzcd}$$

where e.g. the label "$\{A_3, B_3, C_1\}, c = 0$" corresponds to a marking where thread regions A_3,B_3 and C_1 are mapped to 1, hence processed concurrently, the remaining regions are mapped to 0 and the place representing the channel c is mapped to 0 as well, indicating that there are currently no messages in the channel buffer.

To each node of the state graph CBA analysis variables may be associated to formulate constraints among those. Those states may, for instance, be utilized to model the value of a channel buffer before or after applying channel operations. Note that this kind of information can not, for instance, be associated to individual program points within a thread since without any activity within the local thread the buffer state may change due to concurrent actions.

All those structures, including the set of sync points, the thread regions, the execution net and the program state graph are intermediate results computed as temporary results while analyzing a given program section utilizing our CBA framework. They are not provided as an input. Consequently, data and control flow results covered by the same analysis may influence those structures and vice versa until a steady-state is reached.

Additional analysis may be built upon those structures. For instance, potential dead locks can be identified based on the program state graph. Another analysis may identify *race conditions* by computing the set of memory locations written to in every thread region and search for program state graph nodes listing active thread regions accessing an overlapping set of locations. For instance, in our example thread regions A_3, B_1 and C_2 are all writing to the memory location referenced by the variable a. Since in the obtained state graph A_3 and B_1 as well as A_3 and C_2 are active in the same state a race condition has been identified.

4.4.3 The Constraint Solver

The foundation of the Insieme CBA framework is its constraint solver. Besides covering the algorithm capable of obtaining least fixpoint solutions for sets of constraints, its design also determines the shape of the supported constraints and hence their flexibility and expressiveness for modeling data and control flows.

In Section 4.4.1 a naïve algorithm for solving CBA constraints has been presented while in this section the gradual improvements applied on this algorithm towards the algorithm utilized for the Insieme CBA framework are presented. Those include:

- utilizing *worklists* for reduced run-time complexity
- support for *lazy* generated constraints for reduced run-time complexity
- support for *dynamic dependencies* for increased flexibility
- support for *local restarts* for increased accuracy

Those incremental development steps are covered in detail in the following sub-sections.

Basis: The Naïve Constraint Solver

As covered in the overview section, constraint-based analysis are separating the extraction of constraints from some given input structure, e.g. a CFG, an AST or others, from the actual resolution of the obtained constraints. In the introduced setup constraints are formulas over a set of variables V of the shape

$$g \Rightarrow t \sqsubseteq v$$

where $v \in V$ is a variable, g is a monotone *guard* predicate over the variables V and t is an monotone term forming a lower boundary for the value to be assigned to V according to the $\sqsubseteq$ relation associated to the property space of v. A special case are constraints of the form

$$t \sqsubseteq v$$

which are equivalent to

$$true \Rightarrow t \sqsubseteq v$$

Examples of constraints have been provided on page 195.

Algorithm 4.1 A naïve Constraint Solver

Input: C . . . set of constraints of shape $g \Rightarrow t \sqsubseteq v$
Output: A . . . a least fixpoint assignment satisfying all constraints in C

```
// Step 1: init A with the property spaces' ⊥ values
$A := \epsilon$
for $v \in V$ do
    $A := A[v \mapsto \bot(v)]$
end for

// Step 2: gradually fix unsatisfied constraints
while $\exists (g \Rightarrow t \sqsubseteq v) \in C \,.\, A(g) \wedge (A(t) \not\sqsubseteq A[v])$ do
    $A := A[v \mapsto (A[v] \sqcup A(t))]$
end while
```

An assignment A mapping values to the involved variables satisfies a constraint $g \Rightarrow t \sqsubseteq v$ iff the evaluation of the guard predicate g under A, denoted by $A(g)$, is either *false* or the evaluation of t under A is *less-or-equal* to the value assigned to the variable v according to its property space, which is denoted by $A(t) \sqsubseteq A[v]$.

The naïve algorithm for obtaining a least fixpoint solution in the form of a variable assignment A – as it already has been introduced in the overview section – is summarized by Algorithm 4.1. In a first step the resulting assignment is initialized such that all variables are mapped to the $\bot$ value of their property space (see Definition 4.4). In a second step, unsatisfied constraints are located and the assignment is gradually updated to satisfy more and more constraints until all of them are satisfied. Under the assumption that all guards and lower boundaries are monotone and the property spaces satisfy the *Ascending Chain Condition* the algorithm converges at a least fixpoint solution after a finite number of steps [68].

However, while demonstrating the basic operation of a constraint solver, the naïve approach suffers from weak performance due to the requirement of searching for unsatisfied constraints in every iteration of the convergence step. A deficit that can fortunately be compensated.

Step 1: A Worklist-Based Constraint Solver

Worklist algorithms, as they are well established for constraint solving [68], exploit the fact that not all constraints are depending on variables constraint by any other constraint and the fact that the actual dependencies can be obtained statically. For instance, a constraint

$$\text{true} \in v_1 \Rightarrow v_2 \sqsubseteq v_3$$

constitutes a dependency between the variables v_1,v_2 and v_3 such that whenever either v_1 or v_2 is modified the given constraint needs to be re-evaluated. However, a modification of a distinct variable v_4 would not effect this constraint, which therefore does not need to be re-checked.

A *worklist algorithm* utilizes this relation between constraints to keep track of constraints that are required to be re-checked. It does so by storing those into a *list* which is gradually processed and, in case variables are updated, extended by depending constraints. Once this *worklist* reaches an empty state, all constraints are known to be satisfied by the current assignment.

The design of the actual *worklist* data structure and its operations can have a big influence on the speed of convergence of the resulting algorithm. *FIFO*, *FILO* or concepts focusing on *strongly connected components* of the dependency graph formed by the various constraints may be considered [68]. To provide a generic description we utilize an abstract worklist implementation based on three abstract operators

- empty $\in \mathcal{W}$ . . . a constant representing an empty worklist
- insert : $(\mathcal{W} \times 2^{\mathcal{C}}) \rightarrow \mathcal{W}$ . . . a function extending a given worklist by a set of constraints
- extract : $\mathcal{W} \rightarrow (\mathcal{C} \times \mathcal{W})$ . . . a function computing the next constraint to be processed and the worklist reduced by this constraint

where $\mathcal{W}$ is the set of all worklist instances and $\mathcal{C}$ the set of all constraints.

Algorithm 4.2 utilizes those operators to implement an improved version of a constraint solver algorithm. The first step, the initialization of the resulting assignment A, remains the same. However, in the second step, the worklist is initialized with all constraints, since each constraint needs at least to be considered once, before step 3 is processing constraints according to the order determined by the worklist implementation. Whenever the evaluation of a constraint leads to a modification of a value in the assignment, all depending constraints are inserted to the worklist. Thereby, the term

$$\{(g \Rightarrow t \sqsubseteq x) \in C \mid v \in FV(g) \cup FV(t)\}$$

obtains those dependent constraints based on the free variables (FV) of the *guard* predicates and *lower boundary* terms. Since the actual dependencies are assignment-independent, those are typically obtained statically before reaching the loop of the third step.

An improved variation may replace the static set of depending constraints by a dynamically obtained set as it is computed by Algorithm 4.3.

Algorithm 4.2 A Worklist based Constraint Solver

Input: C . . . set of constraints of shape $g \Rightarrow t \sqsubseteq v$
Output: A . . . a least fixpoint assignment satisfying all constraints in C

```
// Step 1: init A with the property spaces' ⊥ values
$A := \epsilon$
for $v \in V$ do
    $A := A[v \mapsto \bot(v)]$
end for

// Step 2: init worklist
$W := \text{insert}(\text{empty}, C)$

// Step 3: process worklist and gradually fix unsatisfied constraints
while $W \neq \text{empty}$ do
    $((g \Rightarrow t \sqsubseteq v), W) := \text{extract}(W)$
    if $A(g) \wedge A(t) \not\sqsubseteq A[v]$ then
        $A := A[v \mapsto (A[v] \sqcup A(t))]$
        $W := \text{insert}(W, \{(g \Rightarrow t \sqsubseteq x) \in C \mid v \in FV(g) \cup FV(t)\})$
    end if
end while
```

Unlike the static version the function DEP_CNSTR ignores dependencies constituted by lower-boundary terms in case the associated guard predicate is not satisfied. This reduces the number of constrains to be checked while converging to a solution. However, it also increases the effort of obtaining depending constraints since, due to its utilization of the current assignment, constraint dependencies have to be re-computed upon every update of A.

The utilization of worklists are crucial for obtaining solvers achieving acceptable performance. Hence, they are widely utilized in practice. However, they still depend on the full set of constraints being available as an input. While for DFA, utilizing constraints not exhibiting *guard* predicates, all the constraints typically have an influence on the analysis results, in a CBA setup it might easily happen that the vast majority of constraints obtainable from some input code structure may be separated from the constraints influencing desired results by unsatisfied predicates – a situation in particular encountered for context sensitive analysis where the vast majority of representable program point is not reachable. Depending on the input code, the fraction of constraints actually influencing desired analysis results can be easily below 1 : 10.000. The overhead of extracting and processing the remaining, effectless constraints is significantly increasing the execution time

Algorithm 4.3 Depending Constraint Collector Function

Globals:
- A ... current variable assignment
- C ... current set of known constraints

Input:
- v ... a variable for which all depending constraints should be obtained

Returns:
- all known constraints depending on the value of v

```
function DEP_CNSTRS($v$)
    $res := \emptyset$
    for all $c = (g \Rightarrow t \sqsubseteq x) \in C$ do
        if $v \in FV(g) \vee (A(g) \wedge v \in FV(t))$ then
            $res := res \cup \{c\}$                ▷ collect all depending constraints
        end if
    end for
    return $res$
end function
```

of analyses by several orders of magnitude resulting in infeasible analyses. Avoiding those steps is therefore crucial for obtaining a suitable analysis infrastructure. Unfortunately, the decision on whether a constraint is effecting the result is not a static property like constraint dependencies and may hence be made only during the actual convergence process. Consequently, the generation of constraints has to be dynamically incorporated into the solver algorithm, as it is realized by our next development step – the *lazy constraint solver*.

Step 2: A Lazy Constraint Solver

The restructuring of the algorithm to follow a *lazy* approach which is dynamically obtaining constraints upon demand requires to re-design the algorithm's interface. Instead of the full set of constraints C the lazy-algorithms interface accepts two parameters: a set of variables $Q \subset V$ the analysis is targeting and a function $gen : V \to 2^C$ capable of resolving all constraints restraining the value of a given variable. The resulting assignment will map the requested variables Q to the values they are assigned to in a least fixpoint solution of the overall set of constraints.

The resolution of constraints is conducted by the function outlined by Algorithm 4.4. It utilizes the constraint generator function *gen* to obtain constraints for previously unresolved variables, initializes the value those variables are mapped to by A to their property space's $\bot$ value and keeps

Algorithm 4.4 Constraint Resolution Function

Globals:
- *gen* . . . a constraint generator function
- *resolved* . . . set of variables which have been previously resolved
- *A* . . . current variable assignment
- *C* . . . current set of known constraints

Input:
- *vars* . . . a set of variables for which constraints should be resolved

Returns:
- set of new constraints constraining the variables in *vars*

```
function RESOLVE(vars)
    $c$ = gen(vars \ resolved)    ▷ obtain constraints of unresolved variables
    for $v \in$ vars \ resolved do
        $A := A[v \mapsto \bot(v)]$    ▷ initialize new variables in A
    end for
    resolved = resolved ∪ vars    ▷ mark new variables as resolved
    $C = C \cup c$    ▷ collect all constraints
    return $c$    ▷ return new constraints
end function
```

track of the full set of resolved constraints C.

Algorithm 4.5 summarizes the overall *lazy constraint solver* algorithm based on the introduced utilities. In the first step, global values including the resulting assignment A, the set of known constraints C and the set of resolved variables are initialized. As for the worklist-based constraint solver algorithm, the second step is initializing the worklist. However, unlike in the previous case, the full set of constraints is replaced by the constraints targeting the value of the queried variables Q. Furthermore, during the gradual constraint resolution in the last step of the algorithm, additional constraints for referenced variables are resolved on demand. Thereby the influence of guard predicates is considered to keep the number of involved constraints as small as possible.

Step 3: A Lazy Constraint Solver with Dynamic Dependencies

Besides the performance benefits of a lazy constraint solver approach gained by avoiding the creation and processing of a vast majority of constraints derivable from a given input code fragment, a lazy constraint solver furthermore provides the opportunity to introduce additional flexibility in the utilized constraint format.

Algorithm 4.5 A Lazy Constraint Solver

Input:
- Q ... set of variables for which least fixpoint values are desired
- *gen* ... a constraint generation function

Output:
- A ... a least fixpoint assignment for the variables in Q

```
// Step 1: initialization
$A := \epsilon$                                            ▷ init assignment
$C := \emptyset$                                            ▷ init known constraint set
resolved := $\emptyset$                                      ▷ init resolved variables

// Step 2: init worklist
$W :=$ insert(empty,RESOLVE($Q$))                          ▷ init and seed worklist

// Step 3: gradually converge to solution
while $W \neq$ empty do
    $((g \Rightarrow t \sqsubseteq v), W) :=$ extract($W$)             ▷ get next constrain

    $W :=$ insert($W$,RESOLVE($FV(g)$))                    ▷ add guard dependencies
    if $\neg A(g)$ then continue                                ▷ check guard

    $W :=$ insert($W$,RESOLVE($FV(t)$))                 ▷ add boundary dependencies
    if $A[t] \sqsubseteq A(v)$ then continue                         ▷ check boundary

    $A := A[v \mapsto (A[v] \sqcup A(t))]$                         ▷ update assignment
    $W :=$ insert($W$,DEP_CNSTRS($v$)})          ▷ schedule dependent constraints
end while
```

For instance, as has been covered in the introduction to constraint based analysis, the given constraint format does not support quantification (see page 196). In the basic CBA setup a quantified constraint of the form

$$\forall f \in \mathcal{F} \;.\; (f \in F_{10} \Rightarrow \{f(a,b) \mid a \in I_{11} \wedge b \in I_{12}\} \subseteq I_{13})$$

has to be encoded by explicitly enumerating all potential values of f similar to

$$\begin{aligned} + \in F_{10} &\Rightarrow \{a + b \mid a \in I_{11} \wedge b \in I_{12}\} \subseteq I_{13} \\ - \in F_{10} &\Rightarrow \{a - b \mid a \in I_{11} \wedge b \in I_{12}\} \subseteq I_{13} \\ * \in F_{10} &\Rightarrow \{a * b \mid a \in I_{11} \wedge b \in I_{12}\} \subseteq I_{13} \\ &\cdots \end{aligned}$$

extended by constraints of the form

$$l_{f_1} \in F_{10} \Rightarrow I_{\text{lable of body of } f_1 \text{ when called from expression 13}} \subseteq I_{13}$$
$$l_{f_2} \in F_{10} \Rightarrow I_{\text{lable of body of } f_2 \text{ when called from expression 13}} \subseteq I_{13}$$
$$\dots$$

where l_{f_*} are the labels of functions of the input program. Those constraints model the effects of the evaluation of user defined functions and closures. Hence, the corresponding list of constraints may quickly become rather extensive when applying analysis on larger and larger code fragments. However, by introducing support for assignment-dependent constraint dependencies the quantification over the potential functions can be moved into the boundary term to form the constraints

$$\bigcup_{f \in F_{10} \cap L} \{f(a,b) \mid a \in I_{11} \land b \in I_{12}\} \subseteq I_{13}$$
$$\bigcup_{f \in F_{10} \cap F} I_{\text{lable of body of } f \text{ when called from expression 13}} \subseteq I_{13}$$

where in this context L is representing the set of operator literals and F the set of user defined IR functions and closures. While, independently of the assignment A, the set of variables the first constrain is depending on is limited to $\{F_{10}, I_{11}, I_{12}\}$, in the second case the set of variables the boundary term depends on is depending on the value assigned to F_{10}. The more functions it may refer to, the more dependencies to different I_* variables are created.

However, unlike in the former case, where a large number of constraints have to be enumerated to model this behavior, the quantification within the constrain allows to model the same relations utilizing only two constraints. Consequently, support for such constraints is desirable within our framework.

Fortunately, the modifications to Algorithm 4.5 are small to integrate *dynamic constraint dependencies*. Let T be the set of all *guard* and *boundary* terms to be encountered within constraints. By adding an additional function vars such that $\text{vars}(t, A) \in 2^V$ corresponds to the variables referenced by term $t \in T$ under an assignment A this additional feature can be integrated by replacing every call $FV(t)$ by $\text{vars}(t, A)$ for every expression t. In particular, the function obtaining all dependent constraints covered by Algorithm 4.3 is updated to the version described by Algorithm 4.6.

Besides the increased flexibility and reduced number of constraints involved in an analysis, this modification also introduces the foundation for the introduction of variables and constraints based on the values assigned to other variables. For instance, the *execution net* and *program state graph* are values assigned to variables associated to the full program, hence variables representing *global information*. However, variables may be attached

Algorithm 4.6 Dynamic Depending Constraint Collector Function

Globals:
- A . . . current variable assignment
- C . . . current set of known constraints
- vars . . . function resolving variable dependencies of terms

Input:
- v . . . a variable for which all depending constraints should be obtained

Returns:
- all known constraints depending on the value of v

```
function DEP_CNSTRS($v$)
    $res := \emptyset$
    for all $c = (g \Rightarrow t \sqsubseteq x) \in C$ do
        if $v \in \text{vars}(g, A) \lor (A(g) \land v \in \text{vars}(t, A))$ then
            $res := res \cup \{c\}$                ▷ collect all depending constraints
        end if
    end for
    return $res$
end function
```

to elements of those structures and constraints may be formulated among those. Therefore, during the course of it evaluation an analysis may process variables based on structures not known at the start of the analysis. Among other consequences, analyses built on top of our CBA framework can be combined into a single set of constraints processed by one instance of the solver algorithm. In particular, preparation steps obtaining the parallel structure of a code fragment and the actual analysis of this structure is not required to be separated into multiple passes as in other approaches [51, 115]. By combining several analyses utilizing a unified single-stage framework provides the opportunity of forwarding information between analyses such that they may mutually benefit from intermediate results. An example situation benefiting from such a combination is the combination of control and data flow equations as has been covered in the overview section introducing the general approach of constraint based analysis.

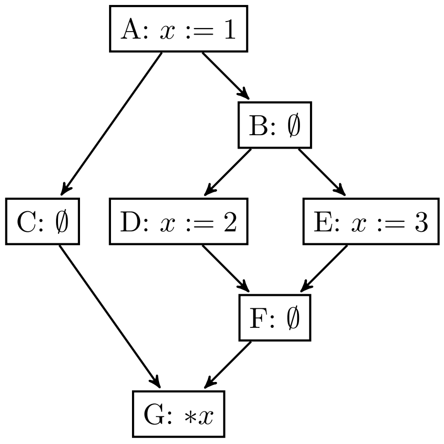
Final Step: Supporting Local Restarts

The last modification developed for the constraint solver algorithm of the Insieme CBA framework is motivated by a more subtle issue encountered when analyzing parallel applications. Consider the following IR code fragment:

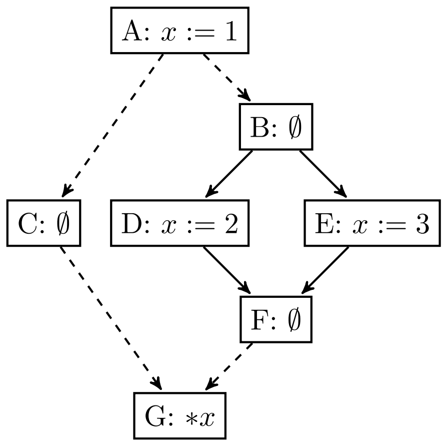
```
auto x = var(int<4>);
x := 1;
merge(parallel(job[1,1]()⇒ {
   if (c) x := 2; else x := 3;
}));
*x;
```

It creates a mutable memory location, initializes it with the value 1, spawns a thread updating the value to 2 or 3 depending on a undetermined condition c, merges the spawned thread group and obtains the value of the location. Clearly, the obtained value should be 2 or 3 since the joining thread has been merged and the inner assignment definitely happened after the initialization. Definitely 1 should not be included in the set of potential values. A static code analysis should be capable of determining this situation.

Parallel Data Flow Analysis An analysis capable of obtaining this result may be design as follows. For each program point and memory location the set of *reachable* and *killed definitions* is obtained. Whenever a value is read, the set of reachable definitions are queried for the assigned values. The foundation of the analysis is given by some (implicit) graph structure similar to



where the nodes are limited to the relevant parts regarding the data stored in the location referenced by the variable x. When applying standard data flow analysis on the given graph structure one would obtain the result that in block G the value of $*x$ is 1, 2 or 3 since this is the result corresponding to the meet-over-all-paths solution obtained by a DFA analysis. However, more accurate results can be obtained when distinguishing sequential and parallel control edges, based on a graph



where dashed edges are parallel edges and solid lines are sequential edges. While diverging paths formed by parallel edges are paths followed by concurrent execution paths, solid edges are optional paths where only one option is to be followed by a sequential thread – e.g. either the *then* or *else* branch of a condition.

For determining the value of x in block G we compute the *killed definitions* and the *reaching definitions* at the begin and end of every block. For *killed definitions* the corresponding constraints for a node x are given by

$$\mathrm{KD_i}[x] \supseteq \bigcup_{y \in \text{p-pred}(x)} \mathrm{KD_o}[y] \cup \bigcap_{y \in \text{s-pred}(x)} \mathrm{KD_o}[y]$$

and

$$\mathrm{KD_o}[x] \supseteq \mathrm{KD_i}[x] \cup \begin{cases} \mathrm{RD_i}[x] & \text{if location is upated in x} \\ \emptyset & \text{otherwise} \end{cases}$$

where $\mathrm{KD_i}[x]$ is the variable describing the killed definition reaching the entry of a block x, $\mathrm{KD_o}[x]$ the killed definitions reaching the end of a block x, $\mathrm{RD_i}[x]$ is the set of reaching definitions at the begin of block x, p-pred(x) lists the predecessors of block x reaching x via a parallel edge and s-pred(x) the predecessors of block x reaching x via a sequential edge. Reaching definitions are defined by the constraints

$$\mathrm{RD_i}[x] \supseteq \left(\bigcup_{y \in \text{p-pred}(x)} \mathrm{RD_o}[y] \cup \bigcup_{y \in \text{s-pred}(x)} \mathrm{RD_o}[y] \right) \setminus \mathrm{KD_i}[x]$$

and

$$\mathrm{RD_o}[x] \supseteq \begin{cases} \{x\} & \text{if location is upated in x} \\ \mathrm{RD_i}[x] & \text{otherwise} \end{cases}$$

where $\mathrm{RD_o}[x]$ represents the definitions reaching the end of block x. Both, the outlined constraints for killed and reaching definitions are regarding a single memory location. In the actual implementation support for multiple locations needs to be added by tracing the definitions for each location individually.

The constraints demonstrate how the different kind of edges are utilized to model the different effects of joining parallel and sequential control flows in the graph illustrated above. While merging parallel control flows which are guaranteed to have happened, only one of the reaching sequential control flow paths is actually processed during the execution of the analyzed code fragment. Hence killed definitions along parallel edges can be aggregated using the union operator while killed definitions along sequential edges have to be intersected.

When solving the given constraints utilizing the algorithms presented so far, the result for $\mathrm{RD}_\mathrm{i}[G]$ could be either $\{A, D, E\}$ or $\{D, E\}$ depending on the order in which constraints are processed – the corresponding computation is left as an exercise. While both are correct solutions, the latter is a more desirable solution due to its higher precision. The reason for the fact that the obtained result is not unique, independent of the order of processed constraints, is the fact that the boundary term provided for $\mathrm{RD}_\mathrm{i}[x]$ is not a monotone expression since the utilized set-difference operation is not monotone. Consequently, since monotonicity is one of the precondition of the algorithms to obtain unique least fixpoints, the produced results do not satisfy those criteria any more.

One option would be to accept the varying quality of solutions. Another would be to replace the constraint by a weaker, yet monotone version. Yet another is to introduce limited support for none-monotone constraints like those into the constraint solver algorithm to ensure that the more accurate solution is always obtained – which is the option we chose for our framework.

Global Monotonicity Essentially monotonicity is required to guarantee the confluence of the full system of constraints. Traditionally this is ensured by restricting all constraints to be monotone. However, this limitation can be relaxed by demanding that the combination of all constrains have to have a monotone effect on the evolving solution instead of a monotone effect on the individual values assigned to analysis variables.

In the traditional interpretation the following restriction has to be valid: Let A_1 and A_2 be two assignments such that for every variable x we have

$$A_1[x] \sqsubseteq A_2[x]$$

Then, by definition, for all constraints $g \Rightarrow t \sqsubseteq v$ we have

$$A_1(g) \Rightarrow A_2(g)$$

and

$$A_1(t) \sqsubseteq A_2(t)$$

due to the *monotonicity* of the predicate g and term t. Hence, the $\sqsubseteq$ relations of the involved property spaces induce a order on the assignments according to which a least fixpoint will be obtained by the constraint solver. However, such an order on variable assignments can still exist even if not all involved constraints are *monotone*. And only this order on the assignments is essential for the existence of a least fixpoint.

Local Restarts In our constraint solver algorithm none-monotone guard and boundary expressions are supported by tracing their values whenever the corresponding constraints are evaluated. As long as in every iteration their value is "larger" than during the last evaluation, the algorithms is behaving as usual. However, in case a term evaluates to a value that is "less" than in the previous iteration, the value assigned to the targeted variable of the currently processed constraint and the assignments of all variables depending on it are "forgotten" by resetting them to the corresponding $\perp$ value iff there is no circular dependency. This modifications introduces support for none-monotone constraints like the set-difference based constraint utilized for the *reaching definitions* analysis outlined above.

The designed solution corresponds to a *trial-and-error* approach where restrictive assumptions may be made, e.g. that no *killed* definitions reach a certain block, and if proven wrong during the course of the analysis, this assumption is dropped and the computation of the depending values is restarted from the beginning. However, if the assumption is not violated more accurate results can be obtained.

To integrate this *trial-and-error* support into our constraint solver algorithm two utility functions are required. The first computes the set of variables depending on a given variable. A naïve version of such an operation is given by Algorithm 4.7. The other required utility is conducting the actual reset of a variable and all elements depending on it and is given by Algorithm 4.8. Based on those utilities the final version of the constraint solver algorithm utilized by the Insieme CBA framework is summarized by Algorithm 4.9.

4.4.4 The Property Space Framework

The aim of the Insieme CBA framework is to provide an infrastructure for the development of static program analyses based on the Insieme IR. To specify an analysis three elements are required:

- a *identifier* for naming associated variables
- a *property space* determining the domain of associated variables and corresponding operators and

Algorithm 4.7 Depending Variable Collector Function

Globals:
- A ... current variable assignment
- C ... current set of known constraints
- vars ... function resolving variable dependencies of terms

Input:
- vs ... variables for which all depending variables should be obtained

Returns:
- all variables depending on elements of vars

```
function DEP_VARS(vs)
    all := $vs$                                          ▷ computes transitive closure
    dep := $\emptyset$                                   ▷ computes depending variables
    repeat
        tmp := dep
        for all $c = (g \Rightarrow t \sqsubseteq x) \in C$ do
            if $\text{all} \cap \text{vars}(g, A) \neq \emptyset \vee (A(g) \wedge \text{all} \cap \text{vars}(t, A) \neq \emptyset)$ then
                all := all $\cup \{x\}$                  ▷ collect transitive closure
                dep := dep $\cup \{x\}$                  ▷ collect all depending variables
            end if
        end for
    until dep = tmp                                      ▷ full closure covered
    return dep
end function
```

- a *constraint generator* to be utilized by the constraint solver to obtain constraints on the associated variables on demand

In previous examples *identifiers* including A, B and F have been utilized for variables associated to arithmetic, boolean or function value analyses. All variables associated to a given analysis have the same *property space* and the constraints on those are created by a single *constraint generator* instance. While the selection of the *identifier* is left completely to the developer of the analysis, for the latter two, generic utilities simplifying the corresponding task are provided.

This section focus on the requirements imposed on *property spaces* to be utilized by the Insieme CBA framework as well as a set of generic utilities provided to handle those. The corresponding tool support for defining *constraint generators* is covered in the following Section 4.4.5.

Algorithm 4.8 Local Reset Procedure

Globals:
– A ... current variable assignment
Input:
– v ... the variable seeding the closure of variables to be reset

```
function RESET($v$)
    $D$ =DEP_VARS($\{v\}$)                ▷ obtains all depending variables
    if $v \notin D$ then                  ▷ only reset if there is no cyclic dependency
        $A := A[v \mapsto \bot(v)]$       ▷ reset value of v
        for all $x \in D$ do
            $A := A[x \mapsto \bot(x)]$   ▷ reset depending variables
        end for
    end if
end function
```

Extended Property Spaces

The foundation of every static program analysis is laid by the design of the associated *property space* $(L, \bigsqcup)$ based on a set of values L and a *combination operator* $\bigsqcup$. A formal definition for those and derived entities is provided by Definition 4.4.

The given definition has been introduced in the context of conventional data flow analysis for sequential programs and inherited by the constraint based analysis approach. Each variable is mapped to an element of L and, in case multiple options are possible, e.g. due to converging control flows, the combination operator $\bigsqcup$ is utilized to obtain the value to be associated to the affected variable.

However, as has been covered in the introduction to parallel data flow analysis on page 225 et seqq., analyses targeting parallel codes may benefit from the distinction of sequential and parallel control flows. Multiple sequential control flow paths reaching or leafing a node in a control flow graph are indicating uncertainty, e.g. introduced by a conditional statement. To the contrast, multiple parallel control flow paths reaching or leaving a node represent a merge or spawn point of concurrent control flows. In the latter case no uncertainty is involved. Hence, for sequential incoming or outgoing paths only one of the potential paths is actually followed during the course of the program execution while in the parallel case all incoming or outgoing paths are followed. This additional information can be exploited by analyses to improve the accuracy of results as has been demonstrated earlier.

To enable analyses to provide different means for merging the effects of sequential and parallel control flow paths we extend the definition of *property*

Algorithm 4.9 A Lazy Constraint Solver Algorithm with Local Restarts

Input:

- Q … set of variables for which least fixpoint values are desired
- *gen* … a constraint generation function
- vars … function resolving variable dependencies of terms

Output:

- A … a least fixpoint assignment for the variables in Q

```
// Step 1: initialization
$A := \epsilon$                                              ▷ init assignment
$C := \emptyset$                                             ▷ init known constraint set
resolved := $\emptyset$                                      ▷ init resolved variables

// Step 2: init worklist
$W$ := insert(empty,RESOLVE($Q$))                            ▷ init and seed worklist

// Step 3: gradually converge to solution
while $W \neq$ empty do
    $((g \Rightarrow t \sqsubseteq v), W)$ := extract($W$)   ▷ get next constrain

    $W$ := insert($W$,RESOLVE($vars(g, A)$))                 ▷ add guard dependencies
    if $\neg A(g)$ and $A(g)$ was valid the last time then RESET($v$)
    if $\neg A(g)$ then continue                             ▷ check guard

    $W$ := insert($W$,RESOLVE($vars(t, A)$))                 ▷ add boundary dependencies
    if last $A[t]$ evaluation $\not\sqsubseteq A[t]$ then RESET($v$)
    if $A[t] \sqsubseteq A(v)$ then continue                 ▷ check boundary

    $A := A[v \mapsto (A[v] \sqcup A(t))]$                   ▷ update assignment
    $W$ := insert($W$,DEP_CNSTRS($v$)})                      ▷ schedule dependent constraints
end while
```

spaces by an additional component.

Definition 4.19 (extended property space)**.** An *extended property space* is given by a triple

$$(L, \bigsqcup, \bigsqcap)$$

where $(L, \bigsqcup)$ is a *property space* and $\bigsqcap : 2^L \to L$ is a second *combination operator* such that $\bigsqcap \emptyset = \bot$ and $\bigsqcap\{l_1, \ldots, l_n, \bot\} = \bigsqcap\{l_1, \ldots, l_n\}$ for all $l_1, \ldots, l_n \in L$. Further, let the binary operator $\sqcap : L \times L \to L$ be defined by $l_1 \sqcap l_2 = \bigsqcap\{l_1, l_2\}$.

Analysis	L	$\bigsqcup$ *one-of-a-set* combinator	$\bigsqcap$ *all-of-a-set* combinator
reaching definitions	set of program points	$\cup$	$\cup$
killed definitions	set of program points	$\cap$	$\cup$

Table 4.1: Examples of Extended Property Spaces.

The first combination operator $\sqcup$ of an extended property space is utilized to merge the effects of optional paths reaching a given program state, e.g. the two branches merged after a conditional statement. The second combination operator $\sqcap$, however, is used to merge the effects of two paths reaching a program point such that both paths have definitely been executed, e.g. when joining threads utilizing a ***merge*** operator call. The $\sqcup$ is hence merging paths where only *one-of-a-set* is actually been followed while $\sqcap$ merges sets of paths where *all* of them have been processed.

Example 4.18 (extended property spaces)**.** Two extended property spaces for well known analyses are outlined by Table 4.1. The property space of *reaching definitions* consist of a set of program points referencing definitions for which the *one-of-a-set* combination operator and the *all-of-a-set* combination operator corresponds to the union operator since if any of the joined parallel or sequential control flows contributes a definition it is to be included in the definitions reaching a program point. To the contrast, an analysis targeting *killed definitions* may utilize set intersection for the *one-of-a-set* combination operator while using a set union for the *all-of-a-set* operator. The former, since it has to assume the most conservative of all the options and the latter since it can be sure that all paths have actually been processed.

Both *combination operators* are utilized by the *generic constraint generators* to be covered in the following section, which provide the foundation for the definition of the corresponding constraint generation components of an analysis. In particular analysis associating variables to program points are benefiting from the *all-of-a-set* combinator (see page 249 et seqq.).

Property Space Constructors for Composed Values

Besides the adaptation of the definition of property spaces for handling combinations of sequential and parallel control flows, designers of property spaces for IR based analyses are confronted by another problem. Let's consider the following code fragment:

```
auto a = struct { x = 10, y = 12 };
```

What is the value of the variable a when applying an arithmetic analysis trying to obtain a set of potential values for expressions? Should it be $\{10\}$, $\{12\}$ or $\{10, 12\}$? Probably the last option, since it summarizes all the values of the included member fields. However, if we extend it to

```
auto a = struct { x = 10, y = 12 };
a.x;
```

we would like the expression $a.x$ to be associate with the value $\{10\}$ only. However, if the variable a is mapped to $\{10, 12\}$, how does the analysis know which value represents the value of the member field x? Also, following this simple approach, the more complex the type of variable a gets, the more inaccurate the projection to member fields becomes.

Similar examples can be provided for union, arrays and vectors. All of those represent constructors for composing values to form more complex structures. In our high-level IR single expressions may represent huge amounts of composed data, organized into hierarchically structured objects of types formed by combining various type constructors.

To the contrast, in a low-level IR, variables represent individual registers of a processor. Hence, each of those may only represent a single machine word, e.g. a integer value, a memory address or a boolean value. In general, no structured data is encountered. However, within the Insieme infrastructure based on its high-level IR the challenge imposed by composed values has to be faced by developers of analyses. The benefits of preserving high-level type information and other structures comes at the price of actually having to deal with those in analyses.

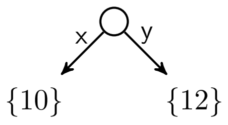
Overview on Modeling Composed Values As has been outlined during the introduction of property spaces, structures wrapping given property spaces can form new property spaces. Examples illustrating the composition of a list of property spaces as well as an example utilizing a given property space as the value-set of a partial mapping have been demonstrated. In the following we will define another, generic way to construct a representation of hierarchical data based on a given property space.

Let $(L, \bigsqcup)$ be an arbitrary property space. The basic idea is to transform it into a property space capable of representing values of hierarchically composed values by constructing a property space where every element is a tree. The leaf notes of those trees are elements of the value set L while the inner nodes resemble the structure of the represented, composed values.

For instance, let $(L, \bigsqcup) = (2^{\mathbb{Z}}, \bigcup)$ be a property space utilized for deducing the arithmetic value of variables. The idea is to present the value of the expression

```
struct { x = 10, y = 12 }
```

by a tree similar to

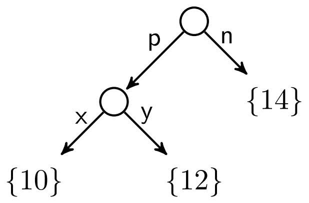


where the root of the tree models the full structure and the edges describe the path to sub-structures. The leaf nodes correspond to values of the original property space.

Consequently, values may also be nested. For instance, the expression

```
struct { p = struct { x = 10, y = 12 }, n = 14 }
```

is represented by the value



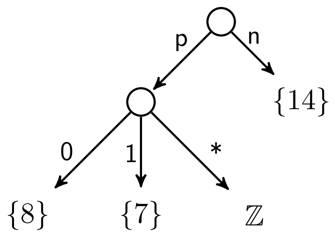
Also, scalar values like the value of the literal *10* can be represented utilizing the special case of a tree

$$\{10\}$$

consisting only of a root note. Further, the descriptive power of the hierarchical composition need not be limited to fields of structs. For instance, the value of the expression

```
struct { p = vector.create([8,7],8), n = 14 }
```

can be modelled by



where the indices 0 and 1 are utilized to address specific elements of a vector and $*$ as an index covering all the remaining elements. Since only the first two of the 8 elements of the constructed array is initialized, the value of the remaining 6 elements remains undefined and needs therefore be modeled by the $\top$ value of the utilized property space which corresponds to $\mathbb{Z}$.

A Property Space Constructor for Composed Values In the following a formal definition for a tree-based property space constructor is provided. In a first step we have to provide a definition for the index sets to be used for labeling edges in the utilized trees.

Definition 4.20 (data index)**.** A *data index* is given by a triple

$$(I, \sqcup, \pi)$$

where I is an arbitrary set, $\sqcup : (2^I \times 2^I) \to 2^I$ a *union operator* for sets of elements of I and $\pi : (2^I \times I) \to 2^I$ a *projection operator*.

Based on this abstract data index definition the tree-based property space constructor is defined as follows:

Definition 4.21 (tree based property space constructor)**.** Let $P = (L, \bigsqcup)$ be a property space and $L_{t(P)}$ be the smallest set such that

- $L \subseteq L_{t(P)}$, hence every element of L is in $L_{t(P)}$ and
- for any data index $I = (I_x, \sqcup_x, \pi_x)$ we have $(I, C) \in L_{t(P)}$ where $C \subseteq (I_x \times L_{t(P)})$

Further, let the *combination operator* $\bigsqcup_{t(P)} : 2^{L_{t(P)}} \to L_{t(P)}$ be defined by

$$\bigsqcup_{t(P)} T = \begin{cases} \bot_P & \text{if } T = \emptyset \\ t_1 \sqcup_{t(P)} \bigsqcup_{t(P)} \{t_2, \ldots, t_n\} & \text{if } T = \{t_1, \ldots, t_n\} \end{cases}$$

where $\bot_P \in L$ is the *bottom element* of the property space P and the binary operator $\sqcup_{t(P)} : (L_{t(P)} \times L_{t(P)}) \to L_{t(P)}$ is given by

$$t_1 \sqcup_{t(P)} t_2 = \begin{cases} t_1 & \text{if } t_2 = \bot_P \\ t_2 & \text{if } t_1 = \bot_P \\ l_1 \sqcup l_2 & \text{if } t_1 = l_1 \in L \text{ and } t_2 = l_2 \in L \\ (I, C_1 \sqcup_{I_x} C_2) & \text{if } t_1 = (I_x, C_1) \text{ and } t_2 = (I_x, C_2) \\ \top_P & \text{otherwise} \end{cases}$$

where the operator $\sqcup_{I_x} : (2^{I_x \times L_{t(P)}} \times 2^{I_x \times L_{t(P)}}) \to 2^{I_x \times L_{t(P)}}$ is given by

$$C_1 \sqcup_{I_x} C_2 = \bigcup_{i \in (I(C_1) \sqcup_x I(C_2))} \left(\bigsqcup_{i_1 \in \pi_x(I(C_1), i)} T(C_1, i_1) \sqcup \bigsqcup_{i_2 \in \pi_x(I(C_2), i)} T(C_2, i_2) \right)$$

where $\sqcup = \sqcup_{t(P)}$, $\sqcup_x$ and π_x are the operators associated to the data index I_x, $I : 2^{I \times L_{t(P)}} \to 2^I$ is defined by

$$I(C) = \{ i \in I \mid \exists t \in L_{t(P)}. (i, t) \in C \}$$

and $T : (2^{I\times L_{t(P)}} \times I) \to L_{t(P)}$ is defined by

$$T(C,i) = \begin{cases} t & \text{if } (i,t) \in C \\ \bot & \text{otherwise} \end{cases}$$

Then the pair $(L_{t(P)}, \bigsqcup_{t(P)})$ is a property space as well.

Every element $t \in L_{t(P)}$ of a tree-value property space derived from a property space P is a tree where every inner node consists of a list of sub-trees indexed by some data index I_x and the leaves are elements of the property space P. The necessary combination operator $\sqcup_{t(P)}$ of the composed property space is merging tree instances on a per-node basis utilizing the union $\sqcup$ and projection operators π of the corresponding data indices as well as the combination operator $\sqcup$ of the underlying property space P. Similarly, a all-of-a-set combination operator $\sqcap_{t(P)}$ can be defined by utilizing the $\sqcap$ operator of the property space P instead, yet its formal introduction has been skipped for brevity. However, tree-value property space constructors can equally be utilized to obtain extended property spaces based on a corresponding extended property space P.

Every node within the tree may utilize an individual data index, providing support for arbitrary compositions of data. For instance, structs may utilize indices based on names while arrays or vectors may utilize some sort of numerical indices. To illustrate their utilization, a few examples shall be provided.

Example 4.19 (nominal data index)**.** Let A be an arbitrary set, e.g. a set of sequences over an alphabet $\mathcal{A}$. A *nominal index* is given by

$$(I_n, \sqcup_n, \pi_n)$$

where the index set I_n is given by

$$I_n = A$$

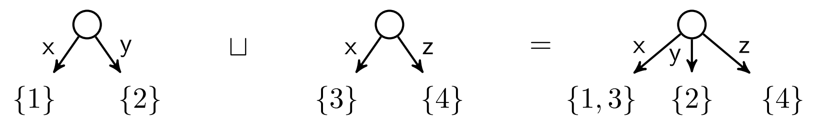
the union operator $\sqcup_n : (2^{I_n} \times 2^{I_n}) \to 2^{I_n}$ is given by

$$\sqcup_n(i_1, i_2) = i_1 \cup i_2$$

and the projection operator $\pi_n : (2^{I_n} \times I_n) \to 2^{I_n}$ is given by

$$\pi_n(i, e) = i \cap \{e\}$$

Hence, to obtain the fraction referenced by an name $e \in I_n$ from a set of partitions addressed by indices $i \subseteq I_n$ the fraction addressed by e is selected if present or none otherwise. An example of the utilization of a nominal index for merging information is given by



where the operator $\sqcup$ is the combination operator of the tree value property space of Definition 4.21. The set of edge labels of the resulting tree value is computed utilizing the $\sqcup_n$ operator. The value referenced by a index element i in the resulting tree is computed by combining the values referenced by the indices $\pi_n(\{x, y\}, i)$ and $\pi_n(\{x, z\}, i)$ in the first and second tree respectively.

Nominal indices based on *identifiers* as index set are utilized in the Insieme CBA framework to model the fields of structs according to the examples provided in the motivation above. However, other structures, including unions, arrays and vectors can not be covered like this. For those other indices are required.

Within unions all fields are mapped to a common data element. Hence, a corresponding data index is required.

Example 4.20 (unit data index)**.** Let unit be an arbitrary token. The unit data index is given by the tuple

$$(I_u, \sqcup_u, \pi_u)$$

where $I_u = \{\text{unit}\}$, the operation $\sqcup_u : (2^{I_u} \times 2^{I_u}) \to 2^{I_u}$ is defined by

$$\sqcup_u(i_1, i_2) = i_1 \cup i_2$$

and the projection operator $\pi_u : (2^{I_u} \times I_u) \to 2^{I_u}$ is given by

$$\pi_u(i, e) = \pi_u(i, \text{unit}) = i$$

For the projection operator, $e \in I_u$ has to be the unit constant, and $i \subseteq I_u$ can only be the empty set $\emptyset$ or $\{\text{unit}\}$. Hence, the list of fragments to be inspected when accessing the element unit in a list of alternatives baring the labels $i \subseteq I_u$ is equivalent to i. Nodes in value trees utilizing the union data index may at most exhibit a single child node reached through a edge labeled unit.

The unit index may also be utilized for modeling arrays – an approach known as *array smashing* where all elements of an array are considered to be a single element [16]. While efficient in terms of memory requirements and computational complexity, more sophisticated indices can increase the precision of analysis. One of those designed for the Insieme CBA framework is the *single data index*.

Example 4.21 (single data index)**.** The *single data index* is given by the tuple

$$(I_s, \sqcup_s, \pi_s)$$

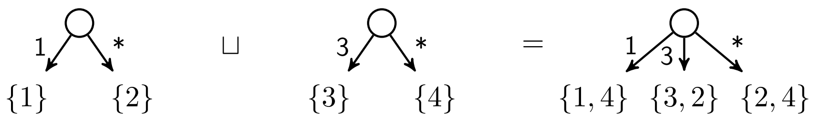
where $I_s = \mathbb{Z} \cup \{*\}$, the union operator $\sqcup_s : (2^{I_s} \times 2^{I_s}) \to 2^{I_s}$ is defined by

$$\sqcup(i_1, i_2) = i_1 \cup i_2$$

and the projection operator $\pi_s : (2^{I_s} \times I_s) \to 2^{I_s}$ is defined by

$$\pi_s(i,e) = \begin{cases} \{e\} & \text{if } e \neq * \wedge e \in i \\ \{*\} & \text{if } e \neq * \wedge e \notin i \\ i & \text{if } e = * \end{cases}$$

For instance, if there are elements referenced by 2, 3 and $*$ and the value associated to index 2 shall be obtained, the projection operator redirects the extraction to the element associated to index 2. If the element 1 should be read, the projection maps the read operation to the element referenced by the $*$ symbol and in case the unknown index $*$ is read all fragments have to be considered, as realized by $\pi_s(\{2,3,*\},*) = \{2,3,*\}$. An example of the utilization of the simple data index is given by



Note that e.g. the value assigned to the index 1 in the resulting tree value is the union of the value 1 is mapped to in the first tree and the value $*$ is mapped to in the second tree. This behavior is specified by the projection function π_s.

The simple data index for arrays and vectors enables the representation of specific values stored within individual elements of an array as well as an aggregated representation of the remaining elements referenced by the $*$ index. However, a problem of the simple data index is that in case of a range of elements sharing a common property the only way to model this is to explicitly enumerate all involved indices. An alternative potentially utilizing less memory and associated computational overhead for those situations is offered by the *range-based data index.*

Example 4.22 (range based data index)**.** Let

$$\mathcal{R} = \{(a,b) \in (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{\infty\}) \mid a \leq b\}$$

be a set of *ranges* such that an element $(a,b) \in \mathcal{R}$ represents the set of integers

$$(a,b) = \{x \in \mathbb{Z} \mid a \leq x < b\} \subseteq \mathbb{Z}$$

For instance, $(3,6) \in \mathcal{R}$ corresponds to the set $\{3,4,5\} \subset \mathbb{Z}$, $(-\infty,2) \in \mathcal{R}$ corresponds to the set $\{x \in \mathbb{Z} \mid x < 2\} = \{\dots,-1,0,1\} \subset \mathbb{Z}$ and $(-\infty,\infty) \in \mathcal{R}$ to $\mathbb{Z}$. Also, for any $x \in \mathbb{Z}$ we have $(x,x) = \emptyset \subset \mathbb{Z}$. The intersection operator $\cap : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ is defined by

$$r_1 \cap r_2 = (a_1,b_1) \cap (a_2,b_2) = (\max(a_1,a_2), \max(\max(a_1,a_2), \min(b_1,b_2)))$$

and extended to sets of ranges by the operator $\cap : 2^{\mathcal{R}} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}$ defined by

$$R_1 \cap R_2 = \{r_1 \cap r_2 \mid r_1 \in R_1 \wedge r_2 \in R_2 \wedge (r_1 \cap r_2) \neq \emptyset\}$$

Also, let the set difference operator $\setminus : \mathcal{R} \times \mathcal{R} \to 2^{\mathcal{R}}$ be given by

$$r_1 \setminus r_2 = (a_1,b_1) \setminus (a_2,b_2) = \begin{cases} \emptyset & \text{if } a_2 \leq a_1 < b_1 \leq b_2 \\ \{(a_1,b_1)\} & \text{if } b_1 \leq a_2 \text{ or } b_2 \leq a_1 \\ \{(a_1,a_2)\} & \text{if } a_1 < a_2 < b_1 \leq b_2 \\ \{(b_2,b_1)\} & \text{if } a_2 \leq a_1 < b_2 < b_1 \\ \{(a_1,a_2),(b_2,b_1)\} & \text{if } a_1 < a_2 < b_2 < b_1 \end{cases}$$

Note that the set $\mathcal{R}$ is not closed under set difference. Hence the result of $r_1 \setminus r_2$ is of type $2^{\mathcal{R}}$. Furthermore, let the $\setminus$ operator be overloaded to allow sets of ranges to be subtracted from a given range by defining $\setminus : \mathcal{R} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}$ as

$$r \setminus R = \begin{cases} r & \text{if } R = \emptyset \\ \bigcup_{s \in r \setminus r_1} s \setminus \{r_2, \dots r_n\} & \text{if } R = \{r_1,\dots,r_n\} \end{cases}$$

and be further extended to sets of ranges by defining the operator $\setminus : 2^{\mathcal{R}} \times 2^{\mathcal{R}} \to 2^{\mathcal{R}}$ by

$$R_1 \setminus R_2 = \bigcup_{r \in R_1} r \setminus R_2$$

The *range based data index* is given by the tuple

$$(I_r, \sqcup_r, \pi_r)$$

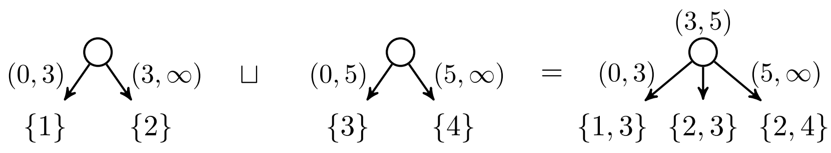
where $I_r = \mathcal{R}$, the union operator $\sqcup_r : (2^{\mathcal{R}} \times 2^{\mathcal{R}}) \to 2^{\mathcal{R}}$ is defined by

$$\sqcup_r(r_1,r_2) = (r_1 \cap r_2) \uplus (r_1 \setminus (r_1 \cap r_2)) \uplus (r_2 \setminus (r_1 \cap r_2))$$

where the operators $\cap$ and $\setminus$ are the operators defined above and the operator $\uplus$ is the union operator for disjoint sets. Finally the projection operator $\pi_r : (2^{\mathcal{R}} \times \mathcal{R}) \to 2^{\mathcal{R}}$ is defined by

$$\pi_r(R,r) = \{s \in R \mid s \cap r \neq \emptyset\}$$

hence, the set of all regions in $R \subseteq \mathcal{R}$ intersecting with the given region $r \in \mathcal{R}$. An example application is illustrated by



which is utilizing the range based data index to model the arithmetic values of e.g. an array instance of unknown length.

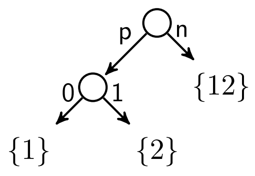
The *range-based data index* demonstrates how advanced index sets can be integrated through the abstract index operator interface comprising $\sqcup$ and π. More complex and specialized indices may be integrated through the given abstract interface, covering e.g. higher-dimensional arrays or high-level data structures like sets or maps.

Finally, to demonstrate the support for modeling more deeply nested values, a concluding example combining structs and vectors within a nested data structure shall be provided.

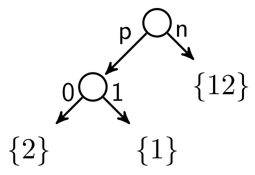
Example 4.23 (heterogeneous composed values)**.** Consider the code fragment

```
auto a =
  ( c ) ?
    struct { p = vector.create([1,2],2), n = 12 }
  :
    struct { p = vector.create([2,1],2), n = 12 };
```

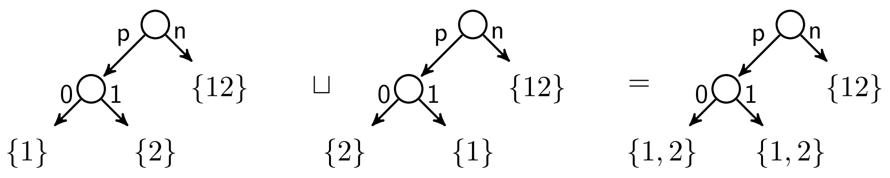
where the variable a is, depending on the value of the condition expression c initialized by one out of two values. The first values is given by



and the second by



Due to the control constraints determined by the involved control flow the value assigned to a is the combination of those two values, assuming c can not be statically determined. Hence, the value of a is given by

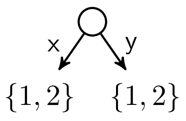


summarizing the most accurate value that can be obtained for all the involved fields.

Preserving Field Relations The utilized tree-based property space constructor is capable of capturing minimal (over-)approximations for the values of the involved fields. However, relations between potential values are not preserved. Consider, for instance, the following example:

```
auto a =
  ( c ) ?
    struct { x = 1; y = 1; }
  :
    struct { x = 2; y = 2; };

auto b = struct { x = 1; y = 2; };

if ( a == b ) { ... }
```

Based on the tree-based property space introduced above, the value of variable a is represented by



from which the values of its fields can be extracted. For instance, the value of $a.x$ is properly obtained to be $\{1, 2\}$. However, the value of a itself may be any of the set $\{(1, 1), (1, 2), (2, 1), (2, 2)\}$. Consequently, when comparing it with the value of b which is fixed to $\{(1, 2)\}$, while analyzing the conditional expression in the last code line, the result will be a *maybe* since neither *true* or *false* can be excluded. However, by inspecting the given code fragment it can be obtained that the condition $a{=}{=}b$ can never be satisfied – independently of the value the expression c evaluates to.

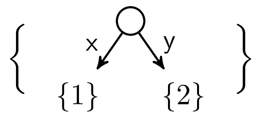
The cause for this over-approximation of the value of the conditional expression is the fact that the utilized tree-based property space is designed to cover the value of the involved fields, yet not their relations. A problem that can be covered by extending the representation of values form individual trees to forests.

Definition 4.22 (forest based property space constructor)**.** Let $P = (L, \bigsqcup)$ be a property space and $L_{t(P)}$ be the derived set of trees as introduced in Definition 4.21 and $L_{f(P)} = 2^{L_{t(P)}}$ be a set of forests. Further, let the *combination operator* $\bigsqcup_{f(P)} : 2^{L_{f(P)}} \to L_{f(P)}$ be defined by

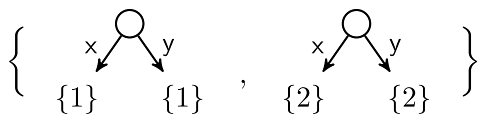
$$\bigsqcup\nolimits_{f(P)} F = \bigcup_{f \in F} f$$

Then the pair $(L_{f(P)}, \bigsqcup_{f(P)})$ is a property space as well.

In a *forest based property space* the value of an expression is represented by a set of trees (= a forest) instead of an individual tree as utilized by a *tree based property space* introduced above. For the example above, the value of the variable b would be represented by



and the value of variable a be represented by



from which can be deduced that the value of a will never be equivalent to the value of b.

Note that that beside the top-level set also the leafs still represent sets of potential values for the involved fields. Those are still required to model uncertain values to be assigned to sub-structures.

Also, the definition of *forest based property spaces* seem simpler than *tree based property spaces* since the index operator $\sqcup$ and π are not utilized. However, when those are still required when extracting the value of fields and elements of a composed structure. Those operations have to be incorporated into the corresponding value constraints generated by the constraint generators feeding the constraint solver algorithm covered in the previous section. The utilities offered by the Insieme CBA framework to support the implementation of constraint generators are covered next.

4.4.5 The Constraint Generator Framework

The Insieme CBA framework provides a generic tool set for the development of analyses based on the Insieme IR. Among others it provides generic implementations of constraint generators – as they are required by the lazy constraint solver – for the various types of variables introduced in the overview section (see page 199). Those included:

- *Labeled Expressions* with call and thread contexts

- *Program Points* addressing states before, during or after the processing of a labeled expression
- *Whole-Program* values covering a property that can not be attributed to a smaller entity and
- *Program State Graph Nodes* in particular for modeling channel states which can not be associated with individual threads

This section provides an overview on those generic constraint generators by outlining the services they are contributing.

Constraint Generators

One of the key components of the CBA framework is a set of generic constraint generators to be customized for specific analysis, as they are required for the lazy constraint solving algorithm developed in Section 4.4.3. The abstract signature of the generator function has been given by

$$gen : V \to 2^C$$

where V is the set of *analysis variables* and C the set of valid constraints. In the framework each variable is specified by a pair

$$(\mathrm{X}, \mathrm{id})$$

which we frequently denote as

$$X_{\mathrm{id}}$$

where X determines the analysis the variable has been introduced by and thus its property space, and the id some structure the variable is associated to, e.g. a label of an expression including its node instance address, call context and thread context or a program program point. For instance, let $l = (i, c, t) \in \mathcal{I}$ be a node label addressing some expression instance and context. Then the variable A_l may denote the *arithmetic value* of the expression addressed by i in call context c and thread t while $R_{l,\mathrm{in}}$ may denote the analysis variable describing whether the program point (l, in) is reachable or not.

The set of available analysis, e.g. A or R, is extendable by designers of analyses utilizing the Insieme CBA framework. For each of those, a property space implementation, including a value type and a combination operator as well as a constraint generator for its associated variables has to be provided. The latter requires the modeling of the semantic of the referenced structures, which in case of labeled expressions and program points is derived from the semantic of the corresponding IR language constructs. Fortunately their treatment is largely identical for all kind of analyses based on the same structure such that a convenient, generic default behavior for those can be specified. Thus, a generic constraint generation implementation providing full, generic default constraint generation capabilities is offered for each kind of supported structure. Those can be inherited by derived analyzes and specialized for their specific use case. This essentially reduces the task of developing a constraint generator for a new analysis to picking a generic generator from the framework and customizing e.g. the treatment of a few literals and operators. Examples of analyses developed that way on top of the Insieme CBA framework are provided in the following Section 4.4.6.

In this section the four basic types of constraint generators, corresponding to the four types of analysis variables, and their responsibilities and example utilizations are outlined.

The Generic Labeled Expression Constraint Generator The first, most frequently utilized constraint generator type is handling the modeling of values expressions may evaluated to during execution. As has been covered in the overview section, *labels* are means to reference instances of expressions by their *node instance address* combined with their call and thread context.

An example is given by the code fragment

```
7 + 4
```

which can be annotate with the labels

$$[[7]^1 + [4]^2]^3$$

where the utilized labels 1,2 and 3 are abbreviations for the actual node instance addresses of the corresponding sub-expressions and their contexts, which in this short example are all identical. Then, for determining potential arithmetic values A_x of the involved labels $x \in \{1, 2, 3\}$ the following constraints are extracted:

$$\begin{aligned} \{7\} &\subseteq A_1 \\ \{4\} &\subseteq A_2 \\ \{x + y \mid x \in A_1 \land y \in A_2\} &\subseteq A_3 \end{aligned}$$

From those constraints the solution $A_3 = \{11\}$ can be obtained. The establishment of those constraints covering the computation of values by expressions is the main responsibility of the *generic labeled expression constraint generator*. However, this task comprises far more than what could be covered by this simple example. The constructs to cover include:

- the interpretation of *literals*,
- the evaluation of function *calls*,
- the support for value *composition* and *access* operators,

- the resolution of the values IR *variables* have been bound to,
- the handling of bind expressions end resulting *closures* and
- the handling of *call contexts* for function calls

The constraints for all those language constructs are generated according to their semantic elaborated in Section 3.7.

Literals The handling of literals has already been demonstrated in the example above. To each literal a value is assigned. By default, the assigned value corresponds to the $\top$ element of the processed property space, indicating that the interpretation of the literal may be anything. Derived analysis, which are typically implemented based on some language extensions (see Section 3.8), intercept this default behavior to introduce specific handling of literals. An example has been shown above by assigning the literals 7 and 4 their corresponding interpretations instead of the default $\top$ value which would, in this example, be equivalent with the full set $\mathbb{Z}$.

Call Expressions The generation of constraints covering the value of call expressions are considerably more complex. On the one hand the targeted function may be a literal, a lambda or a closure. Depending on the actual type, a different treatment is required. On the other hand, the targeted function itself is provided by an expression whose value has to be evaluated by an analysis on its own. An example of this approach has been provided during the introduction of constraint based analysis on page 194 where variables denoted by F_x have been utilized to model the set of targeted functions. Correspondingly, the generic generator for labeled expressions utilizes a specialized instance of itself to compute this set for potential target functions. Based on those, constraints utilizing for-all quantification over the set of callable targets associated to the F_x variables are generated to constrain the value the processed call expression may evaluate to.

In case the targeted function turns out to be a literal, the default behavior is to assume an arbitrary result $\top$ for the value of the call expression. However, as for literals themselves, the interpretation of calls to literals may also be intercepted by derived analyses to model effects of abstract operators. For instance, in the example above, the interpretation of the $+$ operator was intercepted and incorporated into the analysis.

If, however, the target function of the processed call expression is a lambda or closure, the value of the call expression corresponds to the value obtained by evaluating the corresponding function utilizing the present set of arguments. The generic constraint generator therefore creates constraints connecting the value of the processed call expression with the return value of the potentially targeted functions evaluated in a adjusted call context.

Unfortunately the design and implementation details of the associated constraints, the call context handling and thread context handling exceeds the scope of this chapter. For details on those, interested readers may be referred to the actual framework implementation (see Appendix A).

Composition and Access Operators The constraint generator for labeled expressions supports the utilization of *tree* or *forest* based property spaces to model the (composed) value of expressions. For those, *struct* and *union* expressions as well as *array.create* and *vector.create* operations are intercepted and interpreted within the resulting constraints correspondingly. Similarly, access operators, including the *access* expression construct and the *array.subscript* operator are supported as well. While in its current development state the utilized index sets for the various nesting levels are predetermined for each type of nesting level – e.g. a *nominal index* for structs, a *unit index* for unions and a *single data index* for arrays and vectors – this limitation is a mere implementation detail which is planned to be made a configurable option in the future.

Variables The following code fragment illustrates an example where the value of an expression depends on the value of variables:

```
(int<4> b)→int<4> {
  auto a = 5;
  return a + b;
}(12);
```

The value of the expression `a+b` depends on the value of the IR variables `a` and `b`. Let f denote the lambda expression utilized in the given code fragment. Based on the the annotated version

```
[[(int<4> b)→int<4> {
  auto a = [5]¹;
  return [[a]² + [b]³]⁴;
}]⁵([12]⁶)]⁷;
```

the (simplified) constraints

$$\begin{aligned} \{5\} &\subseteq A_1 \\ \{12\} &\subseteq A_6 \\ A_1 &\subseteq A_2 \\ f \in F_5 \Rightarrow A_6 &\subseteq A_3 \\ \{x + y \mid x \in A_2 \wedge y \in A_3\} &\subseteq A_4 \\ f \in F_5 \Rightarrow A_4 &\subseteq A_7 \\ \{f\} &\subseteq F_5 \end{aligned}$$

can be obtained from which the solution $A_7 = \{17\}$ can be derived. The two constraints $A_1 \subseteq A_2$ and $f \in F_5 \Rightarrow A_6 \subseteq A_3$ are incorporating the semantic of variables by linking their usage to their definition points. By doing so, the fact that IR variables are immutable placeholders for values computed at a different code location is utilized. Variables may only be defined in *declaration statements*, *function parameter lists* or as iterators in *for loops*. Depending on their origin, which can be statically deduced from the enclosing IR structure, different sources need to be considered and hence different constraints generated. For *declaration statements* the initialization expression is utilized. For *function parameters* the arguments passed to (all) potential call sites have to be combined and for iterator variables an unknown value $\top$ has to be assumed due to the fact that the corresponding value is re-bound for every iteration. The same treatment is applied to free variables not being declared within the analyzed code fragment. Nevertheless, this default behavior of the generic constraint generator may be overloaded by derived analysis. In particular the handling of loop iterators may be of interest. For instance, for an arithmetic analysis the utilization of a symbolic value for a loop iterator variable may be of interest[2].

A special case is constituted by variables representing parameters bound to arguments captured by bind expressions to form closures. In those cases the evaluation of the arguments may happen in a different call and thread context than the evaluation of unbound parameters. The handling of this relations is essential for modeling the semantic of closures. This none-trivial task is covered as well by the generic generator implementation and is automatically inherited by all derived analyses.

As for the call expressions, the details of those mechanisms are exceeding the scope of this overview chapter. Also, the full details are best presented by the actual implementation (see Appendix A).

Closures and Call Contexts The semantic effects of those two concepts in the Insieme IR are covered implicitly by the handling of call expressions and variables as covered above.

By interpreting literals, call expressions, lambdas and closures, the *generic labeled expression constraint generator* covers the functional core of the Insieme intermediate representation. However, imperative elements, in particular mutable memory locations, require a different kind of constraint generator focusing on program points. Such a generator is provided by the CBA framework and described next.

The Generic Program Point Constraint Generator A program point references the progress of a thread processing a given code fragment. It is defined by a labeled expression or statement (consisting of a node instance

[2]In deed, in the arithmetic analysis, implemented among the standard set of analysis of the Insieme CBA framework, loop iterator variables are treated symbolically.

address, a call context and a thread context) and the phase of its execution (pre, in, post). An analysis based on program points is typically tracing the state of some environment information over the course of the processed instruction stream. Examples include the values stored in a given memory location as well as set of reachable or killed definitions as introduced in the previous section. Also reachability analyses testing whether the control flow can reach a given program point are based on this kind of constraint generator.

The basic task of the generic program point constrain generator is to determine the succeeding or preceding program points of a given point for forward and backward analysis respectively. For instance, when considering the following code fragment

```
(int<4> a)→int<4> {
    return a * 2;
}(2 + 3);
```

and its annotated form

```
[[(int<4> a)→int<4> {
    return [[a]^1 [*]^2 [2]^3]^4;
}]^5 ([[2]^6 [+]^7 [3]^8]^9)]^10;
```

The sequence of program points processed in order by this code fragment is determined by the semantic rules specified in Section 3.7 and given by

$$(10, \text{pre}), (5, \text{pre}), (5, \text{post}), (9, \text{pre}), (7, \text{pre}), (7, \text{post}), (6, \text{pre}), (6, \text{post}),$$
$$(8, \text{pre}), (8, \text{post}), (9, \text{in}), (9, \text{post}), (10, \text{in}), (4, \text{pre}), (2, \text{pre}), (2, \text{post}),$$
$$(1, \text{pre}), (1, \text{post}), (3, \text{pre}), (3, \text{post}), (4, \text{in}), (4, \text{post}), (10, \text{post})$$

The task of the generic *program point constraint generator* is to identify the immediate predecessors / successors of a given point and establish links between the corresponding variables. For instance, when requested to obtain all constraints for a backward program-point-based analysis X regarding the variable $X_{(9,\text{post})}$ the resulting constraint is

$$X_{(9,\text{in})} \sqsubseteq X_{(9,\text{post})}$$

The actual predecessor of a statement may depend on certain conditions. For instance, for the annotated code fragment

```
if ([c]^1) [{
    ...
}]^2 else [{
    ...
}]^3
[s]^4
```

the predecessor of the program point $(4, \text{pre})$ depends on the value of the condition expression c. Those dependencies are encoded by the constraints

$$\begin{aligned} \text{true} \in B_1 &\Rightarrow X_{(2,\text{post})} \sqsubseteq X_{(4,\text{pre})} \\ \text{false} \in B_1 &\Rightarrow X_{(3,\text{post})} \sqsubseteq X_{(4,\text{pre})} \end{aligned}$$

where B_x is determining the boolean value of the labeled expression x. The provided constraints model the conditional control flow. In case c may evaluated to *true* the *then* branch is followed and if c may evaluated to *false* the *else* branch is followed. If the value of the conditional expression c can not be fixed to one of the two values then $B_1 = \{\text{true}, \text{false}\}$, resulting in a case where both branches are followed and joined at the program point $(4, \text{pre})$.

Similar data dependent control flow decisions are encoded for while and *for* loops as well as for call expressions whenever the targeted function is the subject of a dynamic dispatching process.

Note that for simplicity integer values are utilized in the presented examples for labeling expressions. In the actual framework, labels consist of a node instance address, a call context and a thread context. For determining the pre- and successor of a given program point the call and thread contexts have to be considered correspondingly.

Parallel Control Flow At program points causing thread synchronization – hence calls to the spawn, merge, channel and collective operations – parallel control flows are merged. To determine pre- and succeeding program points, those effects have to be incorporated. The necessary information regarding referenced channels and spawned or merged thread bodies is obtained utilizing analyses based on labeled expressions.

Based on this dynamically obtained information, for each program point n the generic program point constraint generator produces a constraint of the form

$$\left(\bigsqcup_{s \in \text{s-pred}(n)} X_s \sqcap \bigsqcap_{p \in \text{p-pred}(n)} X_p \right) \sqsubseteq X_n$$

for backward analysis where X is the specialized analysis identifier, s-pred(n) is the set of predecessors of program point n reached over a sequential control flow step and p-pred(n) is the set of preceding program points reached over a parallel control flow, e.g. a merged thread. Note the combination of the *one-of-a-set* and the *all-of-a-set* combination operators of the underlying extended property space. An example of the utilization of this constraint format has been introduced for the killed definition analysis on page 226. Forward analysis are treated equivalent by substitution predecessors by successors. To introduce conditions on the preceding or succeeding program points the format is further extended to

$$\left(\bigsqcup_{s \in \text{s-pred}(n)} (g_{(s,n)} \Rightarrow X_s) \sqcap \bigsqcap_{p \in \text{p-pred}(n)} (g_{(p,n)} \Rightarrow X_p) \right) \sqsubseteq X_n$$

where $g_{(s,n)}$ and $g_{(p,n)}$ denote predicates determining conditions under which the corresponding predecessor connection is to be considered and $g \Rightarrow X$ is given by

$$(g \Rightarrow X) = \begin{cases} X & \text{if the predicate } g \text{ is satisfied} \\ \bot & \text{otherwise} \end{cases}$$

to introduce conditional dependencies.

Note that this is a consistent extension of the sequential scheme utilized so far since in case of pure sequential edges, where p-pred$(n) = \emptyset$, the second term

$$\bigsqcap_{p \in \text{p-pred}(n)} (g_{(p,n)} \Rightarrow X_p)$$

evaluates to $\bot$, reducing the full constraint on X_n to

$$\bigsqcup_{s \in \text{s-pred}(n)} (g_{(s,n)} \Rightarrow X_s) \sqsubseteq X_n$$

which is equivalent to

$$\begin{aligned} g_{(s_1,n)} \Rightarrow X_{s_1} &\sqsubseteq X_n \\ &\dots \\ g_{(s_k,n)} \Rightarrow X_{s_k} &\sqsubseteq X_n \end{aligned}$$

where s-pred$(n) = \{s_1, \dots, s_k\}$. This version corresponds directly to the input format utilized by the constraint solver and is hence utilized for all the cases not involving parallel control flows – which constitute the vast majority of generated constraints. On the other hand, even the full version of the constraint on X_n presented above fits the input format of the constraint solver when considering the left hand side as the lower boundary. Due to the support of *dynamic dependencies* and *non-monotonic* boundary expressions those constrains can be directly integrated into the system of constraints to be solved by the constraint solver without losing the pruning effect of guard expressions.

Derived analyses built upon the basic setup constituted by the *program point constraint generator* may arbitrarily intercept the resolution of the pre- or successor points. In particular bypassing sequences of program points not effecting the information modeled by the derived analysis can significantly reduce the number of constraints to be processed. Besides those performance improving manipulations, derived analyses are, naturally, required to incorporate the effect of passed operator applications on the modeled state.

The Generic Whole Program Constraint Generator Variables describing properties of the full analyzed code fragment are the easiest to support from the framework's perspective. Since only a single variable is associated to each whole program analysis there is also only a single set of constraints to be computed by the corresponding constraint generator. And this set of constraints is depending on the actual analysis and has therefore to be defined by the derived, concrete analysis. No generic tool support is required to do so.

Examples of whole program analyses include analyses producing structures modeling the parallel execution of the analyzed code fragment. In particular the *execution net* and the *program state graph* (see Definitions 4.16 and 4.18) are incorporated based on whole program analyses. Those also provide the foundation of the last type of variables so far supported by the constraint generation framework.

The Generic Program State Graph Node Constraint Generator In its current development state, the last type of structure supported by the framework for generating constraint on variables is the *program state graph* (see Definition 4.18). Each node of this graph describes a possible phase of the execution of a parallel program involving several threads not processing any synchronizing operations. Invocations of such operations result in edges in the program state graph and constitute transitions between execution phases. An example has been outlined in Example 4.17.

The *generic program state graph node constraint generator* attaches variables to the nodes of the program state graph and connects the values of preceding/succeeding nodes with the values of succeeding/preceding nodes by considering the effects of the linking edges. The interpretation of those edges can be customized by the analysis built on top of this generic constraint generator implementation. By default the *one-of-a-set* combination operator $\bigsqcup$ of the corresponding property space is utilized.

The essential responsibility of this constraint generator is to obtain the *program state graph* in the first place such that it can be utilized for extracting constraints. For this task it relies on a hierarchy of specialized analyses. At the top, this includes a whole program analysis obtaining the actual *pro-*

gram state graph, which relies on a whole program analysis obtaining the *execution net*, which itself is based on a whole-program analysis obtaining the full set of present *sync points*, which relies on a program point based *reaching sync point* analysis, which itself depends on labeled expression analysis covering *jobs*, potential *thread bodies*, *thread groups* and implicitly – due to their labeled expression nature – on labeled expression analyses determining *callables* (=literals, functions or closures) targeted by call expressions, *boolean values* of condition expressions and *arithmetic values* to determine the value of expressions governing the control flow.

As can be seen, the extraction of the program state graph from a given code fragment covers a large variety of analyses and framework components, turning it into an interesting test case for the CBA infrastructure. Also, due to the fact that the graph is the value of an analysis variable, its structure may alter several times before reaching a stable fixpoint solution. This varying nature of the underlying structure has to be handled by the associated generic constraint generator by incorporating corresponding information into the structure utilized to address individual nodes within program state graphs.

Modeling Channel States The main utilization of the program state graph based constraint generators, and the driving force behind its development, is the requirement to model the state of channels, in particular the content of their buffers, over the course of the execution of an IR code fragment. Unlike other states, the state of a channel buffer can not be associated to a single thread by associating it to a program point since even without any progress in a local thread its content may still change due to concurrent operations.

The *program state graph*, on the other hand, provides a proper foundation for the modeling of channel states, as will be demonstrated by the following example.

Example 4.24 (modeling channel states)**.** Consider the following IR code fragment:

```
let int = int<4>;
auto c = channel.create(int,2);
parallel(job[1,1]()⇒ {
  channel.send(c,1);
  channel.send(c,2);
});
int a = channel.recv(c);
parallel(job[1,1]()⇒ {
  channel.send(c,3);
});
int b = channel.recv(c);
```

The program creates a channel, spawns a single thread, waits until a message is received through the channel, binds the value of the message to the variable a, spawns another thread and extracts a second message to be bound to the variable b. The first thread produces and submits two messages while the second sends a single message.

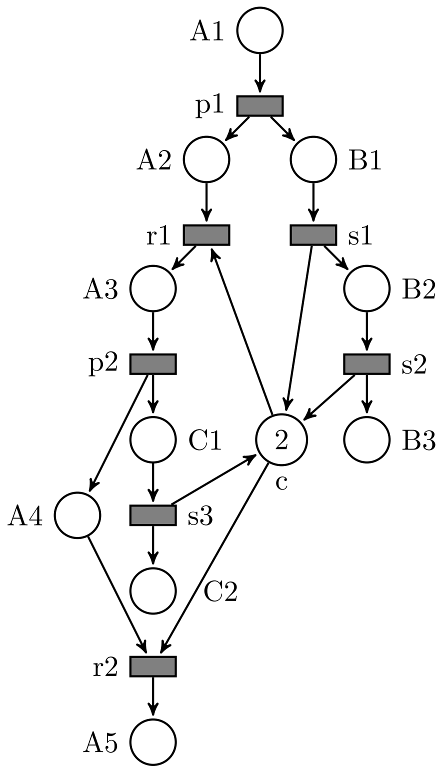
In this example we are interested in the values the variables a and b get bound to. When closely inspecting the given code fragment it can be obtained that the second thread is only started after a first message has been consumed. Hence, a can only get bound to a value produced by the first thread. Since no other *recv* operation is processed before the initialization of a and channels deliver messages in-order, the value a gets bound to is 1. For b it depends on the interleaving of the involved operations whether it will be bound to the value 2 or 3. However, it will not be 1.

Note that those results are obtained by the human reader by reasoning over the parallel structure of the given code fragment. In the Insieme CBA framework support for similar deduction is offered, as will be outlined next.

In a conventional DFA framework, based on low-level instructions unaware of the parallel nature of the processed application, no restrictions on the values of a and b can be obtained since they are the result of a function calls, resulting in a potential value range of $\top = \mathbb{Z} \in 2^{\mathbb{Z}} = L$. An improved approach followed by related work is to produce a more restricted super-set of the potential values obtained by a *recv* operation by computing the union of all the values submitted to a given channel anywhere in the application code [34] ignoring associated synchronization effects [46]. This advanced approach may reduce the set of potential values to $\{1, 2, 3\} \subset \mathbb{Z}$ for both variables in our example. While already providing much higher precision than a conventional approach, our CBA framework, taking the parallel structure of the application into account, can yield even better results.

In our framework the parallel structure of a code fragment is modeled utilizing an *execution net*. It is obtained from the input code by computing the set of synchronization points (begin/end of threads, spawn, merge, channel and collective operations) by combining the results of a variety of label and program point based analyses.

In the given examples we have the begin and end of the three involved threads, the three send operations, $s1$, $s2$ and $s3$, and the two receive operations $r1$ and $r2$ forming the full set of synchronization points. Based on those, the main thread is partitioned into five thread regions $A1$ - $A5$, the first spawned thread into three regions $B1$, $B2$ and $B3$ and the third thread into two regions $C1$ and $C2$. The resulting *execution net* is given by



where all places are labeled by the represented *thread region* or channel instance and transitions by the corresponding synchronization point. Note that the place *c* representing the channel has a capacity of 2 corresponding to the buffer size of the represented channel instance.

From the *execution graph* the following *program state graph* can be obtained:

$$\begin{tikzcd}
 & & \boxed{\{A_1\}, |c| = 0} \arrow[d, "\text{p1}"'] & & \\
 & & \boxed{\{A_2, B_1\}, |c| = 0} \arrow[d, "\text{s1}"'] & & \\
 & & \boxed{\{A_2, B_2\}, |c| = 1} \arrow[dl, "\text{r1}"'] \arrow[dr, "\text{s2}"] & & \\
 & \boxed{\{A_3, B_2\}, |c| = 0} \arrow[d, "\text{p2}"'] \arrow[drr, "\text{s2}"] & & \boxed{\{A_2, B_3\}, |c| = 2} \arrow[d, "\text{r1}"'] & \\
 & \boxed{\{A_4, B_2, C_1\}, |c| = 0} \arrow[d, "\text{s3}"'] \arrow[drr, "\text{s2}"] & & \boxed{\{A_3, B_3\}, |c| = 1} \arrow[d, "\text{p2}"'] & \\
 & \boxed{\{A_4, B_2, C_2\}, |c| = 1} \arrow[dl, "\text{r2}"'] \arrow[dr, "\text{s2}"] & & \boxed{\{A_4, B_3, C_1\}, |c| = 1} \arrow[dl, "\text{s3}"'] \arrow[dr, "\text{r2}"] & \\
\boxed{\{A_5, B_2, C_2\}, |c| = 0} \arrow[drr, "\text{s2}"'] & & \boxed{\{A_4, B_3, C_2\}, |c| = 2} \arrow[d, "\text{r2}"'] & & \boxed{\{A_5, B_3, C_1\}, |c| = 0} \arrow[dll, "\text{s3}"] \\
 & & \boxed{\{A_5, B_3, C_2\}, |c| = 1} & &
\end{tikzcd}$$

As has been covered previously, the nodes in the *program state graph* describe phases of the execution of a parallel program by listing a set of *thread regions* being concurrently processed as well as the number of elements stored within the buffers associated to channels.

As has also been remarked earlier, no channel state changes may happen while processing a *thread region* since send and receive operations – the only operations capable of altering channel states – are constituting boundaries of those. Hence, the *program state graph* can be utilized to establish constraints on the state of channel buffers over the course of a program execution.

To demonstrate this, we identify each of the states of the program state graph with a letter A-M to obtain

$$\begin{tikzcd}
 & & \boxed{A} \arrow[d, "p1"'] & & \\
 & & \boxed{B} \arrow[d, "s1"'] & & \\
 & & \boxed{C} \arrow[dl, "r1"'] \arrow[dr, "s2"] & & \\
 & \boxed{D} \arrow[d, "p2"'] \arrow[drr, "s2"] & & \boxed{E} \arrow[d, "r1"'] & \\
 & \boxed{F} \arrow[d, "s3"'] \arrow[drr, "s2"] & & \boxed{G} \arrow[d, "p2"'] & \\
 & \boxed{H} \arrow[dl, "r2"'] \arrow[dr, "s2"] & & \boxed{I} \arrow[dl, "s3"'] \arrow[dr, "r2"] & \\
\boxed{J} \arrow[drr, "s2"'] & & \boxed{K} \arrow[d, "r2"'] & & \boxed{L} \arrow[dll, "s3"] \\
 & & \boxed{M} & &
\end{tikzcd}$$

Based on this structure we conduct our analysis.

However, before we do so, we need to define our property space[3]. To model the state of a channel buffer of capacity n we utilize the property space $(L_c, \bigcup_c)$ where

$$L_c = 2^{\mathbb{Z}^{\leq n}}$$

and $\mathbb{Z}^{\leq n}$ is given by

$$\mathbb{Z}^{\leq n} = \bigcup_{0 \leq i \leq n} \mathbb{Z}^i$$

and $\bigcup_c$ is the set union operator. Hence, the state of a channel buffer of size n is modeled by a set of tuples up to length n.[4] For our example, since the buffer size is limited to 2, we have $L = 2^{\mathbb{Z}^{\leq 2}}$. We further assume that integer values are modeled using the property space $(2^{\mathbb{Z}}, \bigcup)$. Further, let the function push : $(2^{\mathbb{Z}^{\leq 2}} \times 2^{\mathbb{Z}}) \to 2^{\mathbb{Z}^{\leq 2}}$ be defined by

$$\begin{aligned}
\text{push}(S, A) = & \{[x] \mid [] \in S \wedge x \in A\} \cup \\
& \{[x_1, x] \mid [x_1] \in S \wedge x \in A\} \cup \\
& \{[x_2, x] \mid [x_1, x_2] \in S \wedge x \in A\}
\end{aligned}$$

and the function pop : $2^{\mathbb{Z}^{\leq 2}} \to 2^{\mathbb{Z}^{\leq 2}}$ be defined by

$$\text{pop}(S) = \{[] \mid \exists x_1.[x_1] \in S\} \cup \{[x_2] \mid \exists x_1.[x_1, x_2] \in S\}$$

[3]Since there will only be one-of-a-set edges no extended property space is required.

[4]Compare with the semantic of channels on page 115 et seqq.

for modeling the effects of send and receive operations.

Let $g = (N, E)$ be the the graph illustrated above such that $N = \{A, \ldots, M\}$ and E is the set of triples $e = (n, o, m) \in N \times O \times N$ such that $e \in E$ whenever there is an edge from node n to m labeled by an operation $o \in O$. Further, let C_x be the variable describing the channel state of node $x \in \{A, \ldots, M\}$ of the *program state graph* illustrate above and A_{s1}, A_{s2} and A_{s3} be the variables describing the values sent into the channel at the corresponding program points resulting from a labeled expression based arithmetic analysis. Based on those we establish the constraint

$$\{[]\} \subseteq C_A$$

initializing the channel state with an empty buffer state $[] \in \mathbb{Z}^{\leq 2}$, and for all nodes $x \neq A$ the constraints

$$\bigcup_{(i,o,x)\in E} t_o(C_i) \subseteq C_x$$

where the transfer function $t_o : L \to L$ is defined by

$$t_o(S) = \begin{cases} \text{push}(S, A_o) & \text{if o is a send operation} \\ \text{pop}(S) & \text{if o is a recv operation} \\ S & \text{otherwise} \end{cases}$$

and obtain

$$\begin{aligned} \{[]\} &\subseteq C_A \\ C_A &\subseteq C_B \\ \text{push}(C_B, A_{s1}) &\subseteq C_C \\ \text{pop}(C_C) &\subseteq C_D \\ \text{push}(C_C, A_{s2}) &\subseteq C_E \\ C_D &\subseteq C_F \\ \text{push}(C_D, A_{s2}) \cup \text{pop}(C_E) &\subseteq C_G \\ \text{push}(C_F, A_{s3}) &\subseteq C_H \\ \text{push}(C_F, A_{s2}) \cup C_G &\subseteq C_I \\ \text{pop}(C_H) &\subseteq C_J \\ \text{push}(C_H, A_{s2}) \cup \text{push}(C_I, A_{s3}) &\subseteq C_K \\ \text{pop}(C_I) &\subseteq C_L \\ \text{push}(C_J, A_{s2}) \cup \text{pop}(C_K) \cup \text{push}(C_L, A_{s3}) &\subseteq C_M \end{aligned}$$

and the constraints

$$\begin{aligned} \{1\} &\subseteq A_{s1} \\ \{2\} &\subseteq A_{s2} \\ \{3\} &\subseteq A_{s3} \end{aligned}$$

from the arithmetic value analysis. Those constraints can be solved to obtain the solution

A_{s1}	A_{s2}	A_{s3}	C_A	C_B	C_C	C_D	C_E
$\{1\}$	$\{2\}$	$\{3\}$	$\{[]\}$	$\{[]\}$	$\{[1]\}$	$\{[]\}$	$\{[1,2]\}$

C_F	C_G	C_H	C_I	C_J	C_K	C_L	C_M
$\{[]\}$	$\{[2]\}$	$\{[3]\}$	$\{[2]\}$	$\{[]\}$	$\{[3,2],[2,3]\}$	$\{[]\}$	$\{[2],[3]\}$

describing the state of the buffer associated to channel c during the various phases of the program execution.

To obtain the values to be retrieved by a *recv* operation the channel states associated to preceding program states are examined to obtain the desired values. In our example, the operation $r1$ has a single preceding state C with $C_C = \{[1]\}$ resulting in $\{1\}$ being the value bound to the variable a. Similarly, the preceding states of operation $r2$ are $\{H, K, I\}$ with $C_H = \{[3]\}$,$C_K = \{[3,2],[2,3]\}$ and $C_I = \{[2]\}$ resulting in $\{2,3\}$ to characterize the value to be bound to the variable b – as desired.

Certainly, considering the increased precision in the demonstrated example may hardly justify the increased complexity of the overall framework since after all we could only narrow down the set of potential values by one or two elements. However, considering those values to be utilized within condition expressions those might lead to a increased accuracy of depending results. Also, since any kind of value, in particular references, channels and functions, may be forwarded through channels, increased precision regarding relayed data can have a significant impact on the results of analyses. In particular for references, the popular *alias analysis* can be effected by this.

The channel state analysis, as illustrated by the covered example, is a *program state node* based analysis built based on the corresponding generic constrain generation utility. However, while the outlined channel state analysis is limited to integer values of expressions, it demonstrates a pattern which can be followed by all value analysis to trace the flow of data through channels and hence the propagation of analysis information throughout a processed code fragment. It has therefore been implemented itself in a generic way such that it can be instantiated for arbitrary property spaces. Furthermore, this generic channel-state constraint generator has been integrated into the basic labeled expression constraint generator such that generic support for channel operations is provided out-of-the-box for any derived analyses. However, its full integration is still pending (July 2014).

Finally, unlike demonstrated in the simple example above, our framework is not restricted to statically bound channels. Hence, as for functions targeted by call expressions, the channels addressed by a communication operation may be determined by an arbitrary expression of the corresponding type. For instance, in the code fragment

```
channel<bool,1> a = ... some source ...;
channel<bool,1> b = ... some source ...;
channel.recv( (x < 2) ? a : b );
```

the channel accessed by the conducted *recv* operation is dynamically selected based on the value of some expression. Such a case is supported by producing correspondingly predicated constraints. To determine the addressed channels, analyses based on *labeled expressions* are utilized, thereby harnessing the full power and capabilities of the Insieme CBA framework. In particular, this step even enables channels to be transferred through (other) channels to be then utilized for send and receive operations.

While in classical contexts of channels, in particular stream processing, the dynamic selection of channels in send and receive operations may be of limit utility since all channels are anyway addressed statically, the modeling of other program constructs benefits from this capability. For instance, *locks* are modeled within our IR utilizing channels. As are *futures*. Since those are in general first-class citizens in their host languages, corresponding support in the underlying analyses is required – and provided.

4.4.6 Example Value Analyses

After introducing all the generic infrastructure offered by the Insieme CBA framework to implement analyses this section outlines two example analyses implemented using the available utilities. Those two example analyses cover

- the *arithmetic* value of an expression and
- the *boolean* value of an expression

Both of them are based on corresponding language extensions (see Section 3.8). The first obtains a set of potential (symbolic) arithmetic values for a given expression while the second aims on determining whether a given expression would evaluate to *true* or *false*. The latter is of particular importance since it potentially restricts the value of conditional expressions governing the control flow of an execution while the former may be utilized to determine symbolic values of array subscripts for e.g. loop-dependency analyses. Also, the former is required as a foundation for the latter.

Arithmetic Analysis

To specify an analysis on top of the Insieme CBA framework three things have to be specified: a name, a property space and a constraint generator. For our arithmetic analysis we will stick to A as an identifier. The remaining two elements are covered next.

Property Space Throughout this chapter a property space of $(2^{\mathbb{Z}}, \bigcup)$ has been utilized for analysis focusing on integral values. This simple property space has been utilizes for simplicity. Unfortunately it does not satisfy the *ascending chain condition* [68] since there are infinite sequences of $x_0, x_1, \ldots \in 2^{\mathbb{Z}}$ such that $\forall i \in \mathbb{N} \,.\, x_i \subset x_{i+1}$. In practice this bears the risk that a fixpoint computation may follow this sequence without ever terminating. For instance, in the partially annotated code fragment

```
let int = int<4>;
[ref<int> a = var(0)]^1;
while( c ) {
   [a := (*a) + 1]^2;
}
[*a]^3;
```

an analysis focusing on the value of the memory location referenced by the variable a will result in the following (simplified) constraints:

$$\begin{aligned} \{0\} &\subseteq A_{(1,\text{post},l)} \\ A_{(1,\text{post},l)} &\subseteq A_{(2,\text{pre},l)} \\ A_{(2,\text{post},l)} &\subseteq A_{(2,\text{pre},l)} \\ \{x+1 \mid x \in A_{(2,\text{pre},l)}\} &\subseteq A_{(2,\text{post},l)} \\ A_{(2,\text{post},l)} &\subseteq A_{(3,\text{pre},l)} \end{aligned}$$

where l identifies the memory location created in the second line. The full details on how support for mutable state is integrated into the Insieme CBA framework is covered in the following section. Essentially, the value of each memory location is modeled at each program point of the execution utilizing an extended version of a *generic program point constraint generator*. In the example, the value stored in memory location l after processing expression 1 contains at least $\{0\}$. Before processing expression 2 it may exhibit the values it has been initialized with or, since expression 2 is in a loop, the value it had after the last loop iteration. This is covered by the second and third constraint. The fifth constraint incorporates the effect of the assignment expression 2. Final, the value of the variable after the loop corresponds to the value it had after the last assignment operation.

When solving those constraints the following sequence of value assignments will be processed:

Iteration	$A_{(1,\text{post},l)}$	$A_{(2,\text{pre},l)}$	$A_{(2,\text{post},l)}$	$A_{(3,\text{pre},l)}$
init	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{0\}$	$\{0\}$	$\{0,1\}$	$\{0,1\}$
2	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,2\}$
3	$\{0\}$	$\{0,1,2\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
...	...	...	...	...
n	$\{0\}$	$\{0,\ldots,n-1\}$	$\{0,\ldots,n\}$	$\{0,\ldots,n\}$

Obviously this will not lead to a fixpoint in a finite number of steps. Hence, the property space $(2^{\mathbb{Z}}, \bigcup)$ to model the value of integral values of an expression has to be modified to satisfy the *ascending chain condition.* A simple way to achieve this is to instate an upper limit on the number of values within the sets.

Definition 4.23 (limited power-set property space)**.** Let S be an arbitrary set and $n \in \mathbb{N}$ be a natural number determining the maximum cardinality of sets in the resulting property space. Further, let $L_{|S|\leq n}$ be the set

$$\{s \in 2^S \mid |s| \leq n\} \cup \{S\}$$

and $\bigcup_{|S|\leq n} : 2^{L_{|S|\leq n}} \to L_{|S|\leq n}$ be defined by

$$\bigcup_{|S|\leq n} S = \begin{cases} \bigcup S & \textit{if } |\bigcup S| \leq n \\ S & \textit{otherwise} \end{cases}$$

Than $(L_{|S|\leq n}, \bigcup_{|S|\leq n})$ is a property space based on subsets of S limited to a cardinality of n satisfying the *ascending chain condition.*

We can utilize this property space constructor to define a property space for our integer value analysis satisfying the *ascending chain condition.*

Definition 4.24 (integer constant property space)**.** Let $n \in \mathbb{N}$ be a natural number determining the maximum number of potential values to be determined for an expression, $L_{\text{ic-}n} = L_{|\mathbb{Z}|\leq n}$ and $\bigcup_{\text{ic-}n} = \bigcup_{|\mathbb{Z}|\leq n}$.Then

$$(L_{\text{ic-}n}, \bigcup_{\text{ic-}n})$$

is a property space for determining the set of potential integer values of expressions satisfying the *ascending chain condition.*

Note that the well known property space utilized for *constant folding* introduced by Example 4.5 is a special case of the given property space by utilizing $n = 1$. In this case $\bot_c = \emptyset$ and $\top_c = \mathbb{Z}$.

Utilizing e.g. the property space $(L_{\text{ic-}3}, \bigcup_{\text{ic-}3})$ for analyzing the loop-based example above, the solver would iterate through the assignments

Iteration	$A_{(1,\text{post},l)}$	$A_{(2,\text{pre},l)}$	$A_{(2,\text{post},l)}$	$A_{(3,\text{pre},l)}$
init	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\{0\}$	$\{0\}$	$\{0,1\}$	$\{0,1\}$
2	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,2\}$
3	$\{0\}$	$\{0,1,2\}$	$\mathbb{Z}$	$\mathbb{Z}$
4	$\{0\}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$

and obtain a fixpoint after a finite number of steps. The same modification to satisfy the ascending chain condition can be applied to all property spaces based on power-sets of infinite sets. Power-sets of finite sets satisfy the ascending chain condition innately.

Symbolic Values In many cases sets of potential integer values are of little use. To determine loop dependencies in the code fragment

```
let int = int<4>;
for( int i = 0 .. c ) {
  a[i] := a[i+1];
}
```

the integer values determined by our integer value analysis for the subscripts i and $i+1$ correspond to $\top = \mathbb{Z}$ since i is a loop iterator variable for which no preciser value can be obtained. Obviously, for conducting loop dependency analysis our current integer value property space is not suitable. Symbolic terms are required.

For our arithmetic analysis we are utilizing polynomials over a set of symbolic variables to represent the value of expressions.

Definition 4.25 (polynomial)**.** Let X be a set of symbolic indeterminate (variables). A polynomial in m indeterminates is given by

$$\sum_{\vec{i} \in \mathbb{N}_0^m} a_{\vec{i}} \prod_{j=1}^{m} x_j^{i_j}$$

where $a_{\vec{i}} \in \mathbb{Z}$ are the *coefficients* of the *terms* $\prod_{j=1}^{m} x_j^{i_j}$ which are composed of *factors* $x_j^{i_j}$ based on the *indeterminates* $x_1, \ldots, x_m \in X$. The set of all polynomials over a set of indeterminates X is denoted by $Poly(X)$.

Examples of polynomials in a single variable include expressions like

$$x^2 - 3x + 2$$

and linear formulas like $x + 1$ as well as constant values including 3 and 0. Accordingly, constant values are special cases of polynomials. Polynomials in multiple variables include expressions like

$$y^4 - 2xy^2 + 4x^2y - 2x^2 + 10$$

As usual, we omit coefficients with $|a_{\vec{i}}| = 1$, terms with coefficients $a_{\vec{i}} = 0$ and factors with exponents $i_j = 0$.

Polynomials are closed under addition, subtraction, multiplication and composition. For instance, we have

$$(x^2 - 4) + (y^2 - x + 2) = x^2 + y^2 - x - 2$$

and

$$(x^2 - 1) * (y + 2) = x^2y - y + 2x^2 - 2$$

The details of those operations are omitted for brevity and may be obtained from any algebra text book.

We utilize polynomials as the values to be associated to expressions by defining the following property space for our arithmetic analysis.

Definition 4.26 (arithmetic property space)**.** The *arithmetic property space* is given by the limited power-set property space

$$(L_{|Poly(V)|\leq n}, \bigcup_{|Poly(V)|\leq n})$$

where $V \subset \mathcal{L}$ is a subset of the node labels $\mathcal{L}$ (see Definition 4.11) referencing literals, variables utilized as loop iterators or the first occurrence of free variables within a code fragment and their associated call and thread context.

The value assigned by the arithmetic analysis to an expression within a targeted code fragment is hence a limited set of polynomials over a set of indeterminates constituted by all labels referencing literals, loop iterators and free variables of the analyzed code fragment.

Finally, to support composed values like structs or vectors of values, the set based arithmetic property space is used to construct a tree (or forest) based arithmetic property space using the corresponding constructors of Definition 4.21 and Definition 4.22. As a result, assuming the tree based constructor is selected, the finally utilized property space is given by

$$(L_{t(P_a)}, \bigsqcup_{t(P_a)})$$

where $P_a = (L_{|Poly(V)|\leq n}, \bigcup_{|Poly(V)|\leq n})$. Hence, the arithmetic analysis assigns each targeted expression a tree (or forest) whose inner nodes describes the structure of the represented value and whose leaves are cardinality-limited sets of polynomials over indeterminates constituted by node labels.

Example 4.25 (arithmetic property space values)**.** In the code fragment

```
auto a1 = 12;
auto a2 = c + 3;
auto a3 = a1 * 2 + a2;
auto a4 = a3 - a2;
for ( int i = 1 .. 100 ) {
  auto a5 = 2 * i * a2 + i;
}
auto a6 = ( c < 4 ) ? a1 : a2;
auto a7 = struct { x = a3; y = a6; };
```

the arithmetic analysis would obtain the following values for the involved variables:

Variable	Value
a1	$\{12\}$
a2	$\{c+3\}$
a3	$\{c+27\}$
a4	$\{24\}$
a5	$\{2ci+7i\}$
a6	$\{12, c+3\}$
a7	$(x = \{c+27\}, y = \{12, c+3\})$

Here we are utilize the notation $(x = \{c + 27\}, y = \{12, c + 3\})$ to represent the tree value

$$\begin{tikzcd} & \circ \arrow[dl, "x"'] \arrow[dr, "y"] & \\ \{c+27\} & & \{12, c+3\} \end{tikzcd}$$

Note that, since c is a literal (or free variable) and i is a loop iterator variable those values are utilized as indeterminates within the polynomials.

Constraint Generator After defining a proper property space covering the information to be extracted from a program by our analysis one more component is required to complete the analysis specification: a constraint generator.

Since our analysis targets the value expressions get evaluated to within a code fragment, the constraint generator for our analysis is derived from the *generic labeled expression constraint generator*. This generic implementation deals with all matters related to function calls, composed data types, variables and parameter passing, closures and call- and thread contexts. All this can be inherited. The final step to conduct is to specialize the generic handling of some literals and a few functions to model their effects on the analyzed values.

The list of literals and operators whose interpretation has to be adapted is provided by the arithmetic language extension (see page 125 et seqq.). In particular, those include integer literals and arithmetic operators including *int.add* and *uint.mul*. For instance, for each label l referencing a literal

$$\mathit{lit}(12 : \mathit{int}\,\langle 4\rangle)$$

the generic handler would create a conservative constraint similar to

$$\top \sqsubseteq_{t(P_a)} A_l$$

where $\top = \mathit{Poly}(V)$. The specialized analysis intercepts the constraint extraction for this kind of literals and generates

$$\{12\} \sqsubseteq_{t(P_a)} A_l$$

instead. In case l references a literal of shape

$$\mathit{lit}(c : \mathit{int}\,\langle 4\rangle)$$

where c is some identifier not encoding an integral value, a constraint like

$$\{l\} \sqsubseteq_{t(P_a)} A_l$$

is produced, where l is utilized as an indeterminate in the polynomial $l \in Poly(V)$ – assuming l references the first occurrence of the literal c, otherwise the label l' of the corresponding first occurrence. This corresponds to the interpretation of constants as indeterminates in the polynomials of the property space of our analysis. Free variables and loop iterators are treated identically. For labels c referencing function calls targeting e.g. an addition operator the default handling of the generic labeled expression generator would produce

$$\top \sqsubseteq_{t(P_a)} A_c$$

while the specialized arithmetic constraint generator produces

$$\bigcup\nolimits_{t(P_a)} \{\{x + y\} \mid x \in A_a \wedge y \in A_b\} \sqsubseteq_{t(P_a)} A_c$$

where a is the label of the first argument of the call expression c, b the second argument and $+$ the addition operator for polynomials. Similar, other operators of the arithmetic language extension are specialized accordingly.

Note that those rules utilize the fact that tree values of a tree-based property spaces representing scalar values only consist of a root node corresponding to an element of the value set of the underlying property space. In the present case those are sets of polynomials.

That is all that is required in the Insieme CBA framework to build a constraint generator for a new analysis supporting all the IR constructs and features including variables, function calls, parameters, return values, recursions, closures, channels and composed data values including structs, unions, arrays and vectors.

Example Application: Loop Dependency Tests To conclude this section on the arithmetic analysis we outline how it can be utilized for loop dependence tests. Consider the following code fragment where we are interested in the values of the subscripts $x+j$ and $2*y$ to conduct a dependency test.

```
let int = int<4>;

let f = (ref<array<int>> d, int o)→ unit {
  auto x = o + 1;
  for(int j = 0 .. 10) {
    auto y = o + j;
    d[2*y+2] := d[x+j];
  }
};

ref<array<int>> a = ... some source ... ;
for(int i = 0 .. 5) {
  f(a, 2*i+1);
}
```

Let the label l_1 reference the first subscript and the label l_2 addressing the second subscript in their respective call and thread contexts. The arithmetic analysis introduced in this section will yield

$$\begin{aligned} A_{l_1} &= \{4i + 2j + 4\} \\ A_{l_2} &= \{2i + j + 2\} \end{aligned}$$

where i and j are the labels referencing the expression declaring the corresponding iterator variables. Those symbolic term can then be utilized to deduce that a *true*, *false* and output dependency is present in the given loop nest. For instance, the memory location $a[4]$ is written in iteration $(i, j) = (0, 0)$ and read in iteration $(0, 2)$, constituting a true dependency, while e.g. the memory location $a[8]$ is read in iteration $(0, 6)$ and updated in iteration $(1, 0)$, constituting a false dependency. Also, e.g. the memory location $a[10]$ is updated in iteration $(0, 3)$ and $(1, 1)$.

Boolean Analysis

The second analysis to be outlined in this section is the boolean analysis. Based on the value of boolean expressions branches within application codes may be followed or skipped. Consequently, integrating an accurate analysis determining boolean values of expressions can increase the precision of other analyses. As for the previous analysis, a name, a property space and a constraint generator is required. To denote analysis variables describing the boolean value of an expression we utilize the letter B – as it has already been utilized in several examples throughout this chapter.

Property Space The information we would like to obtain for a boolean expression is whether it might evaluate to *true* or *false*. Correspondingly we define the property space

$$P_b = (2^{\mathbb{B}}, \bigcup)$$

where $\mathbb{B} = \{\text{true}, \text{false}\}$ in a first step. Hence, $\bot = \emptyset$ and $\top = \{\text{true}, \text{false}\}$.

Since $\mathbb{B}$ is a finite set, the subset based property space P_b satisfies the ascending chain condition. Thus we do not have to wrap P_b into a limited power-set property space as in the arithmetic case. However, since boolean values may still occur as fields of structs, unions or elements of arrays and vectors we create a tree (or forest) based property space out of P_b and obtain the property space

$$(L_{t(P_b)}, \bigsqcup\nolimits_{t(P_b)})$$

for our boolean analysis. Hence, for every expression targeted by the boolean analysis a tree is obtained where the inner nodes describe the structure of the modeled value and the leaves are given by a value in

$$\{\emptyset, \{\text{true}\}, \{\text{false}\}, \{\text{true}, \text{false}\}\} = 2^{\mathbb{B}}$$

Constraint Generator As for the arithmetic analysis, the constraint generator for the boolean analysis is based on the generic labeled expression constraint generator. Also, it is based on a language extensions comprising literals and operators (see page 123 et seqq.).

The generation of constraints for the two literals *true* and *false* is intercepted and substituted by proper constraints. For instance, while the default handler would generate the constraint

$$\top \sqsubseteq_{t(P_b)} B_l$$

for a label l referencing the *true* literal, the specialized generator produces the constraint

$$\{\text{true}\} \sqsubseteq_{t(P_b)} B_l$$

Unlike the arithmetic operators, which are all abstract operators, all the operators on the type *bool* are derived operators. Hence, those are not required to be intercepted since their interpretation can be obtained by analyzing their definitions. For instance, the operators *bool.and* or *bool.eq* are defined by a function containing a conditional expression. Hence, no special handling for those is required. The framework resolves them properly. So are the operators accepting lazy constructs, including the *bool.land* and *bool.ite* operators. The framework properly handles the lazy evaluation of the involved functions or closures.

At this point the specification of an analysis benefits from derived operators since those do not have to be handled at all. They are handled generically by the system. Consequently, less constructs need to be covered and processed by analyses which reduces the probability of faulty interpretations. Also, newly introduced derived operators can be handled by analyses without updating those.

However, one class of operators still has to be handled. Comparison operators for arithmetic values produce boolean results and are frequently encountered in conditional expressions. To support those, the boolean analysis relies on the results of the arithmetic analysis. For instance, in the code fragment

```
let int = int<4>;
if ( c + 2 < 12 ) {
   ...
}
```

the boolean value of the condition expression depends on the arithmetic values of the operants of the < operator. Those are obtained by the arithmetic analysis in the form of sets of polynomials. To determine the value of a comparison operation we utilize the following observation:

Let $p_1, p_2 \in \mathit{Poly}(V)$ be two polynomials as they are utilized in the property space of the arithmetic analysis. The value p_1 may evaluate to is less than p_2 for all potential values of the involved indeterminates, denoted by $p_1 < p_2$, if

$$p_1 - p_2 < 0$$

This is for sure the case if the polynomial $p_1 - p_2$ is a negative constant. For instance, $(c+2) < (c+3)$ since $(c+2) - (c+3) = -1 < 0$. In this case the boolean value to be assigned by the analysis to the targeted expression is $\{\text{true}\}$. In case $p_1 - p_2$ is zero or a positive constant the analysis assigns $\{\text{false}\}$ to the targeted expression and in any other case $\{\text{true}, \text{false}\} = \top$. Similar rules are implemented for all comparison operators.

Consequently, the boolean analysis can handle numerical symbols introduced by constants, free variables or loop iterators. Similar, equality operators for references and other language extensions have been integrated by utilizing analyses characterizing the values represented by the abstract types of those extensions.

Other Analyses

The boolean and arithmetic analysis demonstrate the integration of abstract and derived types and operators into the analysis framework and how various analyses may interact to benefit from each other. However, besides those, a variety of additional analyses have been implemented on top of the Insieme CBA framework – partially also to realize advanced framework features like the construction of the program state graph. We conclude this section by enumerating the most essential of those:

- C ... *control flow* analysis – determines a set of functions and/or closures targeted by call expressions
- Ch ... *channel analysis* – determines a set of channel references to be utilized by send / recv operations
- DP ... *data paths* – values describing data paths utilized to navigate composed data values (see page 133)
- R ... *references* – values describing memory locations to be potentially referenced by the value expressions evaluate to (see page 130 et seqq.)
- F ... *functions* – obtains a set of potential lambdas, closures or literals an expression may be reduced to without the creation context; a faster alternative to C utilized for pruning constraints
- J ... *jobs* – determines a set of jobs an expression may be reduced to; required to determine which job may be spawned by a *parallel* call
- U ... *uninterpreted symbols* – a generic, type-independent constant propagation analysis based on a *Herbrand structure*

- TB … *thread bodies* – determines the set of bodies potentially spawned by a $parallel$ call
- TG … *thread groups* – traces thread groups being created by $parallel$ calls and consumed by $merge$ calls

In addition to those labeled expression based analyses the following program point based analyses are included in the current infrastructure:

- RE … *reachability* – determines whether a given program can be reached from the start of the analyzed code fragment; in particular utilized to prune the number of generated constraints
- RSP … *reaching sync points* – the set of sync points that may have preceded a given program point; utilized to build execution nets
- $RSWP$ … *reaching spawn points* – the set of spawn points that may have preceded a given program point; utilized to compute the set of threads merged by $merge$ calls merging all spawned thread groups

And finally, the following whole program analyses are offered as a foundation for the realization of framework features as well as future analyses:

- SP … *sync points* – collects a list of all sync-points in a program comprising their labels and contexts; see Definition 4.13
- TR … *thread regions* – lists all thread regions the execution of a code fragment can be separated in; derived from the set of sync points; see Definition 4.14
- TL … *thread list* – a list of threads involved in the processing of a code fragment;
- EN … *execution net* – the execution net describing the parallel structure of a code fragment; see Definition 4.16;
- PS … *program state graph* – the program state graph describing the interaction of thread regions in the execution net; see Definition 4.18

Beside those analyses, support for one crucial language extension is still missing – mutable memory locations. The following section will cover the support for those in detail.

4.4.7 Mutable State Extension

So far all the infrastructure focused on the core of the Insieme IR. This core does not comprise mutable memory locations. IR variables are bound to values that can not be altered afterwards. To model mutable memory locations the *Mutable State* language extension has been introduced (see page 130 et seqq.). An example utilization is given by

```
let int = int<4>;
ref<int> a = var(8);
if (*a < 10) {
  a := 12;
}
*a;
```

where in the second line a memory location is created, initialized and a reference to it assigned to the IR variable a. In the third line the content of the memory location referenced by a is read and utilized for a comparison operation. The fourth line is updating the content of the memory location under the condition that the previous comparison evaluated to true and the sixth line is reading the value from the location after processing the conditional statement.

Unlike any example covered so far, this code fragment utilizes memory locations to forward data throughout the program execution. Values are stored into memory locations at some program points and obtained or updated at others. Tracing values stored in memory locations is an essential requirement when analyzing IR code fragments. It is also a capability covered generically be the Insieme CBA framework. And it is the topic of this section.

The foundation of this section is laid by the *Mutable State* extension, the included abstract data type $ref\langle\alpha\rangle$, the operators defined on top of it and the semantic specification determining their behavior.

To support mutable memory locations solutions for the following problems have to be provided:

- means for addressing memory locations
- tracing values referencing memory locations
- tracing the values stored in memory locations
- handling those in a parallel context

Within this section solutions for those problems will be elaborated.

Addressing Memory Locations

As a first step, to be capable of incorporating the values stored in memory locations into analyses, means to address them are required. Since memory locations, as channels or threads, are created over the course of the execution of a code fragment and forwarded as first-class citizens, they can not be addressed by any node in the IR tree itself. For instance, in the code fragment

```
let int = int<4>;
ref<int> a = var(8);
ref<int> b = a;
b := 12;
```

the IR variable a and b are two different variables, yet both reference the same memory location. The write operation in the last line effects this location such that after its execution the result of $*a$ and $*b$ would be 12. Thus, addressing memory locations by the variables referring to them is no promising approach. Also, memory locations may survive the life time of the variable they get first bound to. For instance, in the code fragment

```
let int = int<4>;
let f = () → ref<int> {
  ref<int> r = new(0);
  r := .. some procedure ..;
  return r;
}
ref<int> a = f();
```

a memory location created on the heap is bound to the variable r, updated and the reference to the location is returned to the call site where it is bound to the variable a. Now a references the memory location and r does not exist any more.

Within the Insieme CBA framework memory locations are addressed by the label of the expression they have been created by.

Definition 4.27 (memory location addresses)**.** In the Insieme CBA framework a memory location is addressed by the node label $l \in \mathcal{L}$ (see Definition 4.11) referencing the call expression instance creating the memory location. The set of all memory locations is denoted by $\mathcal{M} \subset \mathcal{L}$.

Such a label $l = (i, c, t) \in \mathcal{M}$ contains the node instance $i \in \mathcal{I}$ of the call to the $ref.alloc$ operator – which is the only operator capable of creating memory locations – as well as the call and thread contexts $c \in \mathcal{C}$ and $t \in \mathcal{T}$ the memory location creation has occurred in. This approach constitutes an abstract way of addressing all memory locations accessed and manipulated by a code fragment.

Note that the *var* and *new* operators utilized in the code examples are actually composed operators comprising an internal utilization of the abstract $ref.alloc$ operator (see page 132). Thus they are not required to be treated separately.

Tracing References

In a second step, an analysis capable of tracing references to memory locations is developed. This is the foundation for determining the memory location to be read or written by corresponding operations. Obtaining the memory location a reference is targeting is a value analysis similar to the arithmetic or boolean analysis covered in the previous section. However, unlike utilizing polynomials or truth values as the foundation of the property space, structures modeling memory references are required. One option would be to simply utilize the memory locations directly, hence elements of the set $\mathcal{M}$. However, the *mutable state* extension supports an additional feature: *sub-references* (see page 133)

Sub-references allow to reference nested parts of a memory location, e.g. a field x of the struct stored at position 5 of an array stored in a location $m \in \mathcal{M}$. To provide full support for this feature a corresponding property space is required. As usual, a good start for the development of such a property space is the semantic specification determining the behavior of the feature to be modeled.

Designing a Property Space Definition 3.54 provides a formal definition of the domain of reference values. Each reference value is an element of the set $\mathcal{R} = (\mathcal{L} \times \mathcal{P}) \cup \{\eta\}$ where, in this context, $\mathcal{L}$ is the set of (actual) memory locations, $\mathcal{P}$ a set of data paths and η the constant utilized for the *null* reference. For the analysis an abstract interpretation $\widehat{\mathcal{R}} = (\widehat{\mathcal{L}} \times \widehat{\mathcal{P}}) \cup \{\widehat{\eta}\}$ is required.

For the first component, the abstract memory locations $\widehat{\mathcal{L}}$, the labels used to abstract memory locations can be utilized. Thus, $\widehat{\mathcal{L}} = \mathcal{M}$. For the data path we combine the concrete Definition 3.54 with the abstract data indices introduced by Definition 4.20. While the concrete data path definition contributes the structure to build paths, the abstract data indices provide means to deal with uncertainty along the path. Let $\widehat{\mathcal{P}}$ bet the set of all terms generated by the grammar

$$\begin{aligned} p &::= u \mid d \\ u &::= \bot \mid i.u \\ d &::= \bot \mid d.i \end{aligned}$$

where p is the starting symbol and $i \in I$ an index value of some abstract data index $(I, \sqcup, \pi)$. Finally, $\widehat{\eta}$ is an arbitrary constant such that $\widehat{\eta} \notin (\widehat{\mathcal{L}} \times \widehat{\mathcal{P}})$.

Thus, elements of $\widehat{\mathcal{R}}$ provide abstract means to model the memory locations reference values in an IR code fragment can reference. However, to deal with the inherent uncertainty of static program analyses, sets of such reference values need to be assigned to variables modeling the value of IR references. Furthermore, to support composed values, those sets need to be extended to trees or forests utilizing the corresponding property space constructors of Definitions 4.21 and Definition 4.22.

Definition 4.28 (reference property space)**.** Let $P_r = (2^{\widehat{\mathcal{R}}}, \bigcup)$ be a property space over sets of abstract references. The property space for tracing references in the Insieme CBA framework is given by

$$(L_{t(P_r)}, \bigsqcup_{t(P_r)})$$

based on a tree-value based property space.

Note that since the set of memory locations in a program as well as the nesting level of composed values in a program is finite, the set $\widehat{\mathcal{R}}$ is finite too. Hence, the property space P_r is not required to be a limited power-set property space to satisfy the ascending chain condition.

Obtaining a Constraint Generator As the foundation for the reference analysis the generic labeled expression constraint generator is utilized. It is specialized for the operators $ref.alloc$, $ref.narrow$ and $ref.expand$ – which are the only operations manipulating references (see page 133). To support the computation of the resulting reference values an additional analysis targeting the value of *data paths* is utilized. Its property space is based on trees of subsets of the set of abstract data paths $\widehat{\mathcal{P}}$ and the results obtained from it are used to model the effects of the $ref.narrow$ and $ref.expand$ operators.

Examples and Applications Consider the following code fragment:

```
// some type definitions
let int = int<4>;
let A = struct {
  a : int;        b : vector<int,4>;
};
let B = struct { x : ref<int>; y : ref<int>; };

// demonstrate aliasing variables
ref<int> a = var(8);
ref<int> b = a;

// demonstrating sub-referencing
ref<A> c = var(struct {
  a = 12;
  b = vector.create([1,2],4);
});

ref<int> ca = c.a;
ref<vector<int,4>> cb = c.b;
ref<int> cb2 = c.b[2];

// demonstrating uncertainty
ref<int> d = ( ... ) ? b : cb2 ;

// demonstrating composed values
B e = struct { x = b; y = d; };
```

Let $l_a, l_c \in \mathbb{L}$ be the labels referencing the *ref.alloc* calls within the definitions of the *var* operators utilized for the initialization of the variables a and c. Then the reference analysis obtains the following values for the involved variables:

Variable	Value
a	$\{(l_1, \bot)\}$
b	$\{(l_1, \bot)\}$
c	$\{(l_2, \bot)\}$
ca	$\{(l_2, \bot.a)\}$
cb	$\{(l_2, \bot.b)\}$
cb2	$\{(l_2, \bot.b.2)\}$
d	$\{(l_1, \bot), (l_2, \bot.b.2)\}$
e	$(x = \{(l_1, \bot)\}, y = \{(l_1, \bot), (l_2, \bot.b.2)\})$

Note that the reference analysis, as a side effect, constitutes an alias analysis. For instance, based on the results we obtain that variables a and b as well as the expression $e.x$ are referencing the same memory location, $cb2$ is referencing a fraction of the memory location referenced by cb and $e.y$ may reference the same location as $cb2$. The result also shows that e.g. variables a and $cb2$ are definitely not referencing overlapping memory locations.

Tracing the Value of Memory Locations

The third step is to trace the values stored in the memory locations. Upon creation the values stored in a memory location are undefined. A call to the abstract *ref.assign* operator is the only operation that may alter the state of a memory location and a call to the abstract operator *ref.deref* is the only way to retrieve values.

The basic idea is to follow the stream of program points processed during the course of a program execution for each individual memory location starting from its creation point. If a *ref.assign* operator is encountered which is potentially referencing the investigated memory location or a part of it, the effects of the assignment are incorporated into the maintained value. Also, whenever a *ref.deref* operation is encountered, the currently maintained value is retrieved. In the following the corresponding *location state analysis* utilizing the analysis variable name S is developed.

Property Space Unlike the arithmetic, boolean or reference analysis and similar to the channel state analysis covered so far, the *location state analysis* to be developed in this section is a generic analysis. It describes a pattern to be followed, yet still exposes some parameters to be specialized for specific use cases. In particular the property space is left as a parameter.

For instance, when applying an analysis aiming for determining the arithmetic value of an expression, the property space of the corresponding analysis shall be utilized for modeling the states of all involved memory locations. Similar, when focusing on boolean values, references, data paths, jobs or thread groups the property spaces of the corresponding analysis are utilized. In particular this enables the transition between the values of expressions and the values of memory locations when processing $ref.assign$ and $ref.deref$ operations.

Hence, the utilized property space remains a parameter of this generic analysis. However, for the following development of the constraint generator we introduce the abstract property space

$$P_v = (L_v, \bigsqcup_v)$$

to reference to the property space utilized for describing the values stored in memory locations by an instance of the generic *location state* analysis.

Constraint Generator The foundation for the constraint generator for the *location state* analysis is provided by the generic program point constraint generator. However, we extend the labeling of analysis variables by two additional components. The first covers the type of value we are tracing and the second the memory location it is referring to. For instance, whenever the standard program point constraint generator would utilized a variable

$$S_{l,\mathrm{pre}}$$

where $l \in \mathcal{L}$ is some node label, the customized, yet still generic location state analysis references a variable

$$S_{l,\mathrm{pre},a,m}$$

where a is some analysis, e.g. the arithmetic (A) or boolean (B) analysis, and $m \in \mathcal{M}$ an identifier for the traced memory location.

In addition, the resolution of constraints for variables $S_{l,\mathrm{post},a,m}$ referencing calls to the following operators are intercepted:

- *ref.alloc* ... if the label of the invocation of this operator is equivalent to the label addressing the memory location to be traced, its creation point has been encountered. Hence a simple constraint

  $$\top_v \sqsubseteq_v S_{l,\mathrm{post},a,m}$$

  is created to where $\top_v$ is the top element of the abstract property space P_v. This corresponds to the initialization of them memory location utilizing an undefined value.

- *ref.assign* ... upon encountering an update operation the reference analysis is utilized to determine whether the memory location targeted by the first parameter may reference the memory location m.

If so, the effect of the update operation is incorporated accordingly. This is realized by constraints of the following shape: Let l_1, l_2 be the labels of the two arguments of the *ref.assign* call. Further, let X be the identifier for the variables of analysis a. Then the constraints

$$\bigsqcup_v \left(\bigcup_{d \in \{x | (m,x) \in R_{l_1}\}} \{\text{update}\,(S_{l,\text{in},a,m}, d, X_{l_2})\} \right) \sqsubseteq_v S_{l,\text{post},a,m}$$

and

$$|R_{l_1}| > 1 \Rightarrow S_{l,\text{in},a,m} \sqsubseteq_v S_{l,\text{post},a,m}$$

are generated where R_{l_1} is the analysis variable representing the set of potentially targeted memory location fractions, X_{l_2} the value to be assigned and update : $(L_v, \widehat{\mathcal{P}}, L_v) \to L_v$ a function updating the tree or forest based value passed as the first parameter by exchanging the sub-tree(s) addressed by the data path given by the second parameter by the value of the third parameter. The second constraint is only enabled in case the targeted reference can not be uniquely determined. In this case it has to be assumed that the update is not addressing the traced memory location and may hence not affect its value. This is modeled by forwarding the state after processing the arguments yet before processing the assignment operation ($S_{l,\text{in},a,m}$) to the state after processing the assignment ($S_{l,\text{post},a,m}$).

No additional operators or constants have to be considered. However, for performance reasons the resolution of proceeding program points is specialized. For once, whenever resolving a predecessor of a label addressing a program point before the creation point $m \in \mathcal{M} \subset \mathcal{L}$ the resolution is skipped and a constraint assigning $\top_v$ to the corresponding analysis variable is created – since the value stored in a memory location is undefined before its creation. Also, various techniques to skip sequences of program points not including *ref.assign* calls are employed. Thus, while the basic generic program point constraint generator would follow the program point sequence

$$\ldots, p_n, p_{n+1}, \ldots, p_{n+k-1}, p_{n+k}, \ldots$$

and it can be determined that none of the program points $p_{n+1}, \ldots, p_{n+k-1}$ is referencing a call to a *ref.assign* operator those are skipped and the *location state* constraint generator shortcuts the sequence of program points by following

$$\ldots, p_n, p_{n+k}, \ldots$$

which reduces the number of variables and constraints to be processed by the constraint solver.

Integration into the CBA Framework Finally, to enable a wide utilization of the *location state* analysis it is integrated into the basic generic labeled expression constraint generator. Whenever resolving the value of a call expression labeled by $l \in \mathcal{L}$ targeting the abstract operator $ref.deref$ for an analysis X with a tree or forest based property space $P_x = (L_x, \bigsqcup_x)$ a constraint of the form

$$\bigsqcup_x \left(\bigcup_{(m,d) \in R_{l_1}} \{\text{project}\,(S_{l,\text{in},x,m}, d)\} \right) \sqsubseteq_x X_l$$

is generated where R_{l_1} is the analysis variable covering the set of potential references addressed by the $ref.deref$ operation and project : $(L_x, \widehat{\mathcal{P}}) \to L_x$ is a function extracting the value addressed by the second parameter from the tree / forest value passed as the first argument.

This way the connection between memory locations and labeled expression based value analyses is integrated generically. All value analyses based on labeled expressions are implicitly extended with the capability of supporting the tracing of values of memory locations. Thus, this capability is supported out-of-the-box by the constraint generators offered by the Insieme CBA framework.

In particular, also the reference analysis covered above benefits from this effect. References may be stored in memory locations or sub-structures of memory locations and retrieved accordingly. Pointers, modeled by memory locations storing reference values correspond to this kind of use case. By closing the loop between value and location state analyses, the previously presented alias analysis has been extended to arbitrarily nested pointers.

Examples To demonstrate the tracing of the value of memory locations a few examples shall be provided.

A Simple Write/Read Case Consider the following code fragment

```
let int = int<4>;
ref<int> a = var(int);
a := 1;
a := *a + 2;
*a;
```

which is an abbreviated version of

```
let int = int<4>;
ref<int> a = (type<'a> t) → ref<'a> {
  return ref.alloc('a, memloc.stack);
})(int);
ref.assign(a,1);
ref.assign(a, int.add(ref.deref(a),2));
ref.deref(a);
```

and its labeled version

```
let int = int<4>;
[ref<int> [a]^1 = [(type<'a> t) → ref<'a> {
  return [ref.alloc('a, memloc.stack)]^2;
})(int)]^3;
[ref.assign([a]^4 ,[1]^5)]^6;
[ref.assign([a]^7, [int.add([ref.deref([a]^8)]^9 ,[2]^10)]^11)]^12;
[ref.deref([a]^13)]^14;
```

where the labels for some expressions, e.g. all function literals, have been omitted for brevity. The goal is to obtain the value of the last expression $*a$ utilizing the arithmetic analysis covered in the previous section.

The following constraints are generated – omitting details covering dynamic dispatching issues and constraints not affecting the requested value and hence skipped by the lazy-solver. First, the constraint generator of the arithmetic analysis creates the constraints

$$\begin{aligned}
\{1\} &\sqsubseteq_{t(P_a)} A_5 \\
\bigsqcup\nolimits_{t(P_a)} \left(\bigcup_{(m,d)\in R_8} \{\text{project}\,(S_{9,\text{in},a,m}, d)\} \right) &\sqsubseteq_{t(P_a)} A_9 \\
\{2\} &\sqsubseteq_{t(P_a)} A_{10} \\
\{x+y \mid x \in A_9 \wedge y \in A_{10}\} &\sqsubseteq_{t(P_a)} A_{11} \\
\bigsqcup\nolimits_{t(P_a)} \left(\bigcup_{(m,d)\in R_{13}} \{\text{project}\,(S_{14,\text{in},a,m}, d)\} \right) &\sqsubseteq_{t(P_a)} A_{14}
\end{aligned}$$

where the constraints on A_5 and A_{10} are based on the interpretation of literals, the constraint on A_{11} is based on the effect of an arithmetic operation and the constraints on A_9 and A_{14} are created by the generic interpretation of $ref.deref$ operations. Those constraints reference the variables R_8, R_{13}, $S_{9,\text{in},a,m}$ and $S_{14,\text{in},a,m}$. Hence, constraints for those need to be obtained as well. The reference analysis yields the constraints

$$\begin{aligned}
R_3 &\sqsubseteq_{t(P_r)} R_1 \\
\{(2,\bot)\} &\sqsubseteq_{t(P_r)} R_2 \\
R_2 &\sqsubseteq_{t(P_r)} R_3 \\
R_1 &\sqsubseteq_{t(P_r)} R_4 \\
R_1 &\sqsubseteq_{t(P_r)} R_7 \\
R_1 &\sqsubseteq_{t(P_r)} R_8 \\
R_1 &\sqsubseteq_{t(P_r)} R_{13}
\end{aligned}$$

where the constraints on R_1 are based on the initialization of the IR variable a, R_2 is based on the interpretation of the $ref.alloc$ call, R_3 is based on the generic handling of a call expression (omitting dynamic dispatching issues) and the remaining are based on the handling of IR variables. Finally, the constraint generator of the location state analysis yields the constraints

$$\begin{aligned}
\top_{t(P_a)} &\sqsubseteq_{t(P_a)} S_{2,\mathrm{post},a,2}\\
S_{2,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{3,\mathrm{post},a,2}\\
S_{3,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{6,\mathrm{pre},a,2}\\
S_{6,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{4,\mathrm{pre},a,2}\\
S_{4,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{4,\mathrm{post},a,2}\\
S_{4,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{5,\mathrm{pre},a,2}\\
S_{5,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{5,\mathrm{post},a,2}\\
S_{5,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{6,\mathrm{in},a,2}\\
\bigsqcup_{t(P_a)}\left(\bigcup_{d\in\{x|(m,x)\in R_4\}}\{\mathrm{update}\,(S_{6,\mathrm{in},a,2},d,A_5)\}\right) &\sqsubseteq_{t(P_a)} S_{6,\mathrm{post},a,2}\\
|R_4|>1\Rightarrow S_{6,\mathrm{in},a,2} &\sqsubseteq_{t(P_a)} S_{6,\mathrm{post},a,2}\\
S_{6,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{12,\mathrm{pre},a,2}\\
S_{12,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{7,\mathrm{pre},a,2}\\
S_{7,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{7,\mathrm{post},a,2}\\
S_{7,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{11,\mathrm{pre},a,2}\\
S_{11,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{11,\mathrm{post},a,2}\\
S_{11,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{12,\mathrm{in},a,2}\\
\bigsqcup_{t(P_a)}\left(\bigcup_{d\in\{x|(m,x)\in R_7\}}\{\mathrm{update}\,(S_{12,\mathrm{in},a,2},d,A_{11})\}\right) &\sqsubseteq_{t(P_a)} S_{12,\mathrm{post},a,2}\\
|R_7|>1\Rightarrow S_{12,\mathrm{in},a,2} &\sqsubseteq_{t(P_a)} S_{12,\mathrm{post},a,2}\\
S_{12,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{14,\mathrm{pre},a,2}\\
S_{14,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{13,\mathrm{pre},a,2}\\
S_{13,\mathrm{pre},a,2} &\sqsubseteq_{t(P_a)} S_{13,\mathrm{post},a,2}\\
S_{13,\mathrm{post},a,2} &\sqsubseteq_{t(P_a)} S_{14,\mathrm{in},a,2}
\end{aligned}$$

where the constraint for $S_{2,\mathrm{post},a,2}$ is based on the interception of the creation point, the constraints on $S_{6,\mathrm{post},a,2}$ and $S_{12,\mathrm{post},a,2}$ are based on the interpretation of the assignment operator and the rest is produced by the generic program point constraint generator. Note that between the program point $(11,\mathrm{pre})$ and $(11,\mathrm{post})$ several steps have been skipped by the location state constraint generator since no assignment operation can be encountered while processing expression 11. Also note that, as usual, the numerical labels are utilized in this example for simplicity and real labels are combinations of node instance addresses, call- and thread-contexts.

Algorithm 4.9 is gradually obtaining those constraints and computing a smallest fixpoint assignment ass for the involved variables. Among others, the assignment contains the values

$$\begin{aligned}
\text{ass}[S_{2,\text{post},a,2}] &= \top_{t(P_a)} \\
\text{ass}[S_{6,\text{in},a,2}] &= \top_{t(P_a)} \\
\text{ass}[S_{6,\text{post},a,2}] &= \{1\} \\
\text{ass}[S_{12,\text{in},a,2}] &= \{1\} \\
\text{ass}[S_{12,\text{post},a,2}] &= \{3\} \\
\text{ass}[R_4] &= \{(2,\bot)\} \\
\text{ass}[R_7] &= \{(2,\bot)\} \\
\text{ass}[A_9] &= \{1\} \\
\text{ass}[A_{10}] &= \{2\} \\
\text{ass}[A_{11}] &= \{3\} \\
\text{ass}[A_{14}] &= \{3\}
\end{aligned}$$

where $\text{ass}[A_{14}] = \{3\}$ corresponds to the desired result of our analysis.

The presented example demonstrates the interaction between the various analyses and generic constraint generation utilities to obtain values for expressions depending on data and control flow covering functional IR core features and mutable memory locations introduced by a language extension.

Multiple Analysis and Multiple Memory Locations The fact that variables referencing memory locations have been extended by two additional subscript elements – an analysis and a memory location – in addition to the program point it is referring to enables the system to handle an arbitrary number of locations and analyses. Furthermore, by representing the values of memory locations based on tree or forest values, data structures may be arbitrarily nested. For instance, consider the following code fragment:

```
let int = int<4>;
let point = struct { x : int; y : int };
let mark = struct { p : point; b : bool; };

ref<mark> a = var(mark);
ref<mark> b = var(mark);
ref<mark> c = ( .. some condition .. ) ? a : b;

a.p.x := 1;
b.p := struct { x = 0; y = 2 };
c.b := true;
```

In this code fragment two memory locations are created. The first is referenced by the IR variable a and the second by the IR variable b. The variable c may reference the first or the second memory location. Let $m_1, m_2 \in \mathcal{M} \subset \mathcal{L}$ be the labels addressing those memory locations, $l_c \in \mathcal{L}$ be the label of the declaration statement of the IR variable c and $l_{a1}, l_{a2}, l_{a3} \in \mathcal{L}$ be the labels of the three assignments. After the declaration block the value of those is undefined. Hence, for any assignment ass obtained by an analysis instance it follows that

$$\begin{aligned}
\text{ass}[S_{l_c,\text{post},a,m_1}] &= \top_{t(P_a)} \\
\text{ass}[S_{l_c,\text{post},b,m_1}] &= \top_{t(P_b)} \\
\text{ass}[S_{l_c,\text{post},r,m_1}] &= \top_{t(P_r)} \\
\text{ass}[S_{l_c,\text{post},a,m_2}] &= \top_{t(P_a)} \\
\text{ass}[S_{l_c,\text{post},b,m_2}] &= \top_{t(P_b)} \\
\text{ass}[S_{l_c,\text{post},r,m_2}] &= \top_{t(P_r)}
\end{aligned}$$

where a,b and r are tokens to identify and distinguish arithmetic, boolean and reference analysis.

The reference targeted by the first assignment is given by $(m_1, \bot.p.x) \in \widehat{\mathcal{R}}$. Hence, it references a sub-structure of the memory location m_1 only. After the assignment operation the states have altered to

$$\begin{aligned}
\text{ass}[S_{l_{a1},\text{post},a,m_1}] &= (p = (x = \{1\})) \\
\text{ass}[S_{l_{a1},\text{post},b,m_1}] &= (p = (x = \{\text{true}\})) \\
\text{ass}[S_{l_{a1},\text{post},r,m_1}] &= (p = (x = \top_{P_r})) \\
\text{ass}[S_{l_{a1},\text{post},a,m_2}] &= \top_{t(P_a)} \\
\text{ass}[S_{l_{a1},\text{post},b,m_2}] &= \top_{t(P_b)} \\
\text{ass}[S_{l_{a1},\text{post},r,m_2}] &= \top_{t(P_r)}
\end{aligned}$$

Note that, while the arithmetic and the boolean analyses are capable of providing an interpretation for the literal 1, the reference analysis is unable to associate a valid interpretation and therefore defaults to the value $\top_{P_r}$. After the second assignment targeting the location referenced by

$$\text{ass}[R_{l_{b.p}}] = \{(m_2, \bot.p)\}$$

the state

$$\begin{aligned}
\text{ass}[S_{l_{a2},\text{post},a,m_1}] &= (p = (x = \{1\}))\\
\text{ass}[S_{l_{a2},\text{post},b,m_1}] &= (p = (x = \{\text{true}\}))\\
\text{ass}[S_{l_{a2},\text{post},r,m_1}] &= (p = (x = \top_{P_r}))\\
\text{ass}[S_{l_{a2},\text{post},a,m_2}] &= (p = (x = \{0\}, y = \{2\}))\\
\text{ass}[S_{l_{a2},\text{post},b,m_2}] &= (p = (x = \{\text{false}\}, y = \{\text{true}\}))\\
\text{ass}[S_{l_{a2},\text{post},r,m_2}] &= (p = (x = \top_{P_r}, y = \top_{P_r}))
\end{aligned}$$

is reached and after the last assignment targeting

$$\text{ass}[R_{l_{c.b}}] = \{(m_1, \bot.b), (m_2, \bot.b)\}$$

the state

$$\begin{aligned}
\text{ass}[S_{l_{a2},\text{post},a,m_1}] &= (p = (x = \{1\}), b = \top_{P_a})\\
\text{ass}[S_{l_{a2},\text{post},b,m_1}] &= (p = (x = \{\text{true}\}), b = \{\text{true}, \text{false}\})\\
\text{ass}[S_{l_{a2},\text{post},r,m_1}] &= (p = (x = \top_{P_r}), b = \top_{P_r})\\
\text{ass}[S_{l_{a2},\text{post},a,m_2}] &= (p = (x = \{0\}, y = \{2\}), b = \top_{P_a})\\
\text{ass}[S_{l_{a2},\text{post},b,m_2}] &= (p = (x = \{\text{false}\}, y = \{\text{true}\}), b = \{\text{true}, \text{false}\})\\
\text{ass}[S_{l_{a2},\text{post},r,m_2}] &= (p = (x = \top_{P_r}, y = \top_{P_r}), b = \top_{P_r})
\end{aligned}$$

is reached. Note that since c is not uniquely identifying a single potential target the value of addressed sub-structures of the referenced memory locations can not be overridden but have to be merged with the previous values. However, since no explicit values have been present before, the previous values all correspond to the $\top$ elements of the various property spaces.

Nested Pointers and Alias Analysis In a final, short example, the support for pointer and associated pointer alias analysis shall be covered. Consider the following code fragment

```
let int = int<4>;

ref<int> a = var(1);              // a scalar
ref<int> b = var(2);              // another scalar
ref<int> c = a;                   // an alias of a
ref<ref<int>> p = var(b);         // a pointer on b

*a;          // = { 1 }
*b;          // = { 2 }
*c;          // = { 1 }
**p;         // = { 2 }
```

```
a := 3;

*a;          // = { 3 }
*b;          // = { 2 }
*c;          // = { 3 }
**p;         // = { 2 }

p := c;

*a;          // = { 3 }
*b;          // = { 2 }
*c;          // = { 3 }
**p;         // = { 3 }

*p := 4;

*a;          // = { 4 }
*b;          // = { 2 }
*c;          // = { 4 }
**p;         // = { 4 }
```

The comments outline the results that are obtained when applying the arithmetic analysis on the associated expressions. Since pointers are mere memory locations containing references of other memory locations, and since all value analyses are implicitly extended to memory locations, arbitrarily nested pointers can be accurately analyzed using the given infrastructure. No additional or explicit treatment of those is required.

Parallel Control Flow

So far, memory locations are properly handled for sequential control flows. In this fourth, final step, support for parallel control flows is integrated. The basic idea is to integrate the affects of parallel control flows on the state of memory locations by tracing lists of *reaching* and *killed* definitions (see page 225 et seqq.).

Reaching and Killed Definition Analysis A definition is referenced by the node instance address, call and thread context of the expression conducting it, hence a label $l \in \mathcal{L}$ referencing a call expression targeting the abstract $ref.assign$ operator.

Definition 4.29 (reaching definition property space)**.** Let $\mathcal{D} \subset \mathcal{L}$ denote the set of definition points. Then

$$P_{RD} = (2^{\mathcal{D}}, \bigcup, \bigcup)$$

is the extend property space for the *reaching definition* analysis.

Definition 4.30 (killed definition property space)**.** Let $\mathcal{D} \subset \mathcal{L}$ denote the set of definition points. Then

$$P_{KD} = (2^{\mathcal{D}}, \bigcap, \bigcup)$$

is the extend property space for the *killed definition* analysis.

Since those analyzes are not targeting the value of expression in a code fragment, their property spaces are not required to be wrapped into a tree structure. Also, since the number of definition points is finite, no limitation on the cardinality of the utilized sets is required to satisfy the ascending chain condition.

Constraint Generators For both analyzes constraints are generated by specialized versions of the generic program point constraint generator. The associated variables are identified by the names RD and KD and their subscript is extended by the targeted memory location. For instance, the analysis variable

$$RD_{l,\text{in},m}$$

determines the set of definitions potentially targeting the memory location m reaching the program point (l, in). Thus, for every memory location, individual reaching and killed definitions are computed.

Furthermore calls targeting the abstract operator $ref.assign$ are intercepted. Let $l \in \mathcal{D} \subset \mathcal{L}$ be the label of such a call expression and $l_1 \in \mathcal{L}$ the label of the first argument targeting the referenced memory location. Then for a memory location $m \in \mathcal{M}$ the reaching definition analysis generates the constraints

$$\begin{gathered}\exists d \,.\, (m, d) \in R_{l_1} \Rightarrow \{l\} \subseteq RD_{l,\text{post},m} \\ |R_{l_1} \setminus \{(m, \perp)\}| > 1 \Rightarrow RD_{l,\text{in},m} \subseteq RD_{l,\text{post},m}\end{gathered}$$

and the killed definition analysis produces the two non-monotonic constraints

$$\begin{aligned}&R_{l_1} \neq \{(m, \perp)\} \Rightarrow KD_{l,\text{in},m} \supseteq KD_{l,\text{post},m} \\ &R_{l_1} = \{(m, \perp)\} \Rightarrow KD_{l,\text{in},m} \cup RD_{l,\text{in},m} \supseteq KD_{l,\text{post},m}\end{aligned}$$

In all cases the set of potentially referenced memory locations R_{l_1} is inspected to determine how the processing of the assignment might affect the sets of reaching and killed definitions regarding the memory location m. For instance, if the assignment target is $\{(m, \perp)\}$, hence it can be proven that the reference is targeting the memory location m, then all former definitions are killed and the current assignment becomes the only reaching definition for the subsequent program points. However, if the targeted memory location is uncertain, e.g. by having $R_{l_1} = \{(m, \perp), (m', \perp)\}$ for some $m' \neq m$, then the possibility that this assignment might not kill former definitions has to be considered. Correspondingly, former definitions may reach subsequent program points and the definitions reaching the assignment might not get killed. The constraints provided above cover this behavior.

Note that since for the property space of the killed definition analysis the intersection operator $\bigcap$ is utilized as the one-of-a-set combination operator, the abstract operator $\sqsubseteq$ corresponds to $\supseteq$, $\bot = \mathcal{D}$ and $\top = \emptyset$.

The other abstract operator that has to be considered are merge operators. While killed definitions are properly aggregated by the generic implementation of the underlying constraint generator by computing the union of the killed definitions of all merged threads using the *all-of-a-set* combination operator, the handling of reaching definitions has to be further adapted. While the generic generator would create a constraint similar to

$$\left(\bigcup_{(l,s)\in\{\text{end point of joined thread}\}} RD_{l,s,m} \right) \subseteq RD_{l_m,\text{post},m}$$

where $l_m \in \mathcal{L}$ is the label of the merge operation, the reaching definition constraint generator is customized to produce

$$\left(\bigcup_{(l,s)\in\{\text{end point of joined thread}\}} RD_{l,s,m} \right) \setminus KD_{l_m,\text{post},m} \subseteq RD_{l_m,\text{post},m}$$

instead. Hence, all reaching points are collected and reduced by the set of definitely killed definitions contributed by the joined threads. Note that this is another non-monotonic constraint.

Finally, to reduce the number of analysis variables and constraints, for any program point $(l, p) \in \mathcal{P}$ preceding the creation point of the memory location m constraints of the shape

$$\emptyset \subseteq RD_{l,p,m}$$
$$\emptyset \supseteq KD_{l,p,m}$$

are created. Also, as for the location state analysis, steps along sequences of program points not including assignment operations are skipped.

Integration into the Location State Analysis To integrate the effects of parallel control flows on the memory locations the reaching definitions at each merge point are utilized to compute the state of each memory location after the merge.

For each memory location m, analysis X with property space $P_x = (L_x, \bigsqcup_x)$ and program point $(l, \text{post}) \in \mathcal{P}$ referencing a point after a call to a merge operator, the location state generator produces a constraint

$$\bigsqcup_x \{S_{d,\text{post},x,m} \mid d \in RD_{l,\text{post},m}\} \sqsubseteq_x S_{l,\text{post},x,m}$$

combining the values of all reaching definitions at the merge point. This way the affects of relevant assignment operations are combined at the merge point.

A more refined approach may furthermore respect the sub-structure of the handled memory location targeted by the definition points to increase the accuracy when updating independent sub-structures in concurrent threads. However, the necessary details are omitted for simplicity.

Example To demonstrate the handling of parallel control flows and their affects on the values associated to memory locations consider the following code fragment:

```
let int = int<4>;
ref<int> x = var(int);
x := 1;
merge(parallel(job[1,1]()⇒ {
  x := 2;
}));
*x;
```

Clearly, since the contained thread is spawned and merged before the final expression is evaluated, the value of $*x$ is 2.

Based on the (partially) annotated version

```
let int = int<4>;
[ref<int> x = var(int)]^1;
[x := 1]^2;
[merge(parallel(job[1,1]()⇒ {
  [x := 2]^3;
}))]^4;
[*x]^5;
```

the following simplified constraints are obtained by the analysis framework. Let $m \in \mathcal{M}$ be the label addressing the memory location created in the initialization of the variable x. From the arithmetic analysis the constraint

$$S_{5,\mathrm{in},a,m} \sqsubseteq_{t(P_a)} A_5$$

is obtained. It reflects the fact that the value of the expression bearing the label 5 is the result of a read operation targeting memory location m. The details regarding the identification of the addressed memory location by resolving the value of x are omitted for brevity. When resolving constraints for the variable $S_{5,\mathrm{in},a,m}$ the location state constraint generator yields

$$S_{5,\mathrm{pre},a,m} \sqsubseteq_{t(P_a)} S_{5,\mathrm{in},a,m}$$

for $S_{5,\mathrm{pre},a,m}$ the constraint

$$S_{4,\mathrm{post},a,m} \sqsubseteq_{t(P_a)} S_{5,\mathrm{pre},a,m}$$

and for $S_{4,\text{post},a,m}$ the constraint

$$\bigsqcup_x \{S_{d,\text{post},a,m} \mid d \in RD_{4,\text{post},m}\} \sqsubseteq_{t(P_a)} S_{4,\text{post},a,m}$$

merging the values of all reaching definitions. The constraints for reaching definitions are given by

$$\begin{aligned}
\emptyset &\subseteq RD_{1,\text{pre},m}\\
RD_{1,\text{pre},m} &\subseteq RD_{1,\text{post},m}\\
RD_{1,\text{post},m} &\subseteq RD_{2,\text{pre},m}\\
RD_{2,\text{pre},m} &\subseteq RD_{2,\text{in},m}\\
\{2\} &\subseteq RD_{2,\text{post},m}\\
RD_{2,\text{post},m} &\subseteq RD_{3,\text{pre},m}\\
RD_{3,\text{pre},m} &\subseteq RD_{3,\text{in},m}\\
\{3\} &\subseteq RD_{3,\text{post},m}\\
RD_{2,\text{post},m} &\subseteq RD_{4,\text{pre},m}\\
RD_{4,\text{pre},m} &\subseteq RD_{4,\text{in},m}\\
(RD_{4,\text{in},m} \cup RD_{3,\text{post},m}) \setminus KD_{4,\text{post},m} &\subseteq RD_{4,\text{post},m}
\end{aligned}$$

and for killed definitions by

$$\begin{aligned}
\emptyset &\supseteq KD_{1,\text{pre},m}\\
KD_{1,\text{pre},m} &\supseteq KD_{1,\text{post},m}\\
KD_{1,\text{post},m} &\supseteq KD_{2,\text{pre},m}\\
KD_{2,\text{pre},m} &\supseteq KD_{2,\text{in},m}\\
KD_{2,\text{in},m} \cup RD_{2,\text{in},m} &\supseteq KD_{2,\text{post},m}\\
KD_{2,\text{post},m} &\supseteq KD_{3,\text{pre},m}\\
KD_{3,\text{pre},m} &\supseteq KD_{3,\text{in},m}\\
KD_{3,\text{in},m} \cup RD_{3,\text{in},m} &\supseteq KD_{3,\text{post},m}\\
KD_{2,\text{post},m} &\supseteq KD_{4,\text{pre},m}\\
KD_{4,\text{pre},m} &\supseteq KD_{4,\text{in},m}\\
KD_{4,\text{in},m} \cup KD_{3,\text{post},m} &\supseteq KD_{4,\text{post},m}
\end{aligned}$$

respectively. In both cases details dealing with restrictions on the referenced memory locations are omitted for clarity.

From those constraints Algorithm 4.9 computes (with potential local restarts) an assignment ass containing

$$\text{ass}[RD_{4,\text{post},m}] = \{3\}$$

such that, due to the constraint on $S_{4,\text{post},a,m}$, constraints on $S_{3,\text{post},a,m}$ are required. Those are produced by the location state constraint generator resulting in the (simplified) constraint

$$\{2\} \sqsubseteq_{t(P_a)} S_{3,\text{post},a,m}$$

covering the result of the assignment operation bearing the label 3. Based on this constraint Algorithm 4.9 is capable to reach a fixpoint solution ass for the given set of constraints such that

$$\text{ass}[A_5] = \{2\}$$

as desired.

4.4.8 Summary of the Insieme CBA Framework

In this section the Insieme CBA framework has been covered. After providing a basic introduction into program analysis gradually developing the concept of constraint based analysis based on the more widely known data flow analysis an overview on the Insieme CBA framework has been provided in Section 4.4.2. It has been followed by sections elaborating details regarding the underlying constraint solver (4.4.3), utilities offered for constructing property spaces (4.4.4) and the framework provided for the implementation of constraint generators (4.4.5). Furthermore, the design of various analyses built on top of the Insieme CBA framework (4.4.6) and a set of analyses supporting the integration of the mutable state language extension (4.4.7) have been covered in detail.

The resulting framework offers:

- A lazy constraint solver capable of dealing with dynamic constraint dependencies and non-monotonic constraints
- A set of generic utilities to construct property spaces covering arbitrary composed data values based on an extensible set of type constructors; current support covers structs, unions, tuples[5], vectors and arrays;
- A comprehensive set of generic constraint generators based on various types of data structures providing implicit support for all Insieme IR language features and several extensions; Those include: handling of variables, (recursive) functions, closures, dynamic dispatching issues, imperative control flow constructs, jobs, thread groups, concurrent control flows, spawn and merge operations, collective operations and channels; Furthermore support for the arithmetic, boolean, array, vector and mutable state language extensions is included out-of-the-box.
- A collection of basic analyses to be utilized and combined to build advanced analyses and/or analyses based utilities

[5]Tuples have been omitted within this section for brevity but may be considered equivalent to structs with numerical field names.

Altogether those utilities provide a comprehensive toolbox for analyzing IR based parallel codes. The design of new analysis is reduced to the tasks of

- picking a name
- designing a property space
- customizing a generic constraint generator

The full framework has been implemented as part of the Insieme Compiler project [29]. The sources and a comprehensive list of test cases demonstrating various examples are available online (see Appendix A).

4.5 Polyhedral Analyses

The polyhedral model (PM) is a widely utilized and powerful approach for analyzing code sections, in particular focusing on loop nests and loop dependencies [22, 12]. It is based on an algebraic description of a code fragment based on a decidable formalism. Consequently, it imposes restrictions on the supported input codes since in the general case decision problems on program codes are undecidable. However, a large variety of (scientific) codes fulfill this requirement [35].

Due to the restriction to a decidable subset of input codes, (most) analysis based on the polyhedral model yield accurate results. On the contrary, DFA and CFA based approaches covered so far are capable of handling arbitrary input codes, yet necessarily produce over- or under-approximations of the desired information.

4.5.1 Overview on the Polyhedral Model

The polyhedral model has been extensively covered in the literature [22, 12, 35] such that this section just briefly outlines the basic concepts based on an example analysis.

Consider the following code fragment consisting of two nested loops and two array accesses:

```
let int = int<4>;
for(int i = 0 .. N) {              // 0 ≤ i < N
  for(int j = i .. N) {            // i ≤ j < N
    ... = a[i][j-1];               // S1
    a[i][j] := ...;                // S2
  }
}
```

The goal of the example should be to determine whether there are dependencies between various iterations of the loops that might, for instance, prevent those loops from being parallelized.

To obtain this information two steps have to be conducted. In a first step a polyhedral model based description of the given code fragment has to be extracted and in a second step this representation has to be utilized to deduce the desired information.

Polyhedral Model Extraction

In the polyhedral model every statement of a code fragment is represented by a triple consisting of an *iteration domain*, an *affine schedule function* and a list of *access functions*. The first of those, the *iteration domain*, is specified by a convex polytope which has to be defined first.

Definition 4.31 (convex polytope)**.** Let $A \in \mathbb{Z}^{m \times n}$ be an integer matrix and $\vec{b} \in \mathbb{Z}^m$ an integer vector. The set of solutions $\vec{x} \in \mathbb{Z}^n$ of the inequality

$$A\vec{x} + \vec{b} \geq \vec{0}$$

is a n-dimensional *convex polytope.*

Every convex polytope is specified by an integer matrix A and a vector b. However, conventionally the inequation

$$A\vec{x} + \vec{b} \geq \vec{0}$$

is re-written into a homogeneous form

$$\begin{bmatrix} A & \vec{b} \end{bmatrix} \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix} \geq \vec{0}$$

such that a single matrix $A' = \begin{bmatrix} A & \vec{b} \end{bmatrix}$ suffices for the definition of a convex polytope.

Convex polytopes exhibit various useful properties. In particular, the intersection of two convex polytopes, which is equivalent to the conjunction of the defining inequalities, is again a convex polytope and the problem of deciding whether a polytope is empty is NP-hard, and hence decidable.

Iteration Vector The polytopes utilized for modeling the iteration domains of statements in a given code fragment all exhibit the same dimensionality. Furthermore, the interpretation of each dimension is consistent throughout all polytopes. This interpretation is fixed by the so-called *iteration vector*. In the given example the iteration vector $\vec{I}$ is given by

$$\vec{I} = \begin{pmatrix} i & j & N & 1 \end{pmatrix}^T$$

where the first two elements i and j are the loop iterators, N is a *global parameter* and the constant 1 is included to fit the homogeneous form.

Iteration Domain The *Iteration domain* associated with a statement models the individual instances of the corresponding statement within a program execution. In our example, the iterator domain $\mathcal{D}_{S1}$ for the first statement is given by the convex polytope specified by

$$\mathcal{D}_{S1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{bmatrix}$$

which corresponds to the solution set of $\mathcal{D}_{S1}\vec{i} \geq \vec{0}$ which is equivalent to the set of solutions of the constraints

$$\begin{aligned} i &\geq 0 \\ -i + N - 1 &\geq 0 \\ -i + j &\geq 0 \\ -j + N - 1 &\geq 0 \end{aligned}$$

which are equivalent to

$$\begin{aligned} &i \geq 0 \\ &i \leq N - 1 \\ &j \geq i \\ &j \leq N - 1 \end{aligned}$$

which correspond to the boundary conditions of the enclosing loops. Each point in the polytope represents an instance of statement $S1$. For instance, the point $(3, 8, N, 1) \in \mathcal{D}_{S1}$ corresponds to the execution of the statement $S1$ when $i = 3$ and $j = 8$ assuming $N > 8$. Since statement $S2$ is enclosed by the same loops as $S1$, it follows that $\mathcal{D}_{S2} = \mathcal{D}_{S1}$.

Schedule Function The iterator domain models the instances of a statement, yet does not define their execution order. This is covered by the *schedule function.* The schedule function maps each instance of a statement, modeled implicitly by a point of the iterator domain, to a *logical execution time* consisting of an integer vector. The various instances of the involved statements are then processed according to the lexicographical order of those logical timestamps.

Formally, the schedule function is specified for each statement S by an integer matrix $\mathcal{T}_S \in \mathbb{Z}^{k \times n}$, where n is the length of the iteration vector and k the length of the vector representing the logical time, such that fore every statement instance i in the convex polytope defined by $\mathcal{D}_S$ of a statement S the vector $\mathcal{T}_S i$ corresponds to the associated logical timestamp.

In our example, we have to ensure that every instance of statement $S1$ and $S2$ of the loop iteration (x, y) is processed before the corresponding instances of iteration $(x, y+1)$ which themselves are to be processed before the instances of the iteration $(x+1, y)$. Furthermore, every instance of statement $S1$ has to be processed before the corresponding instance of $S2$ of the same loop iteration.

An instance of a statement is given by an element of its associated *iterator domain*. In our example, $(x, y, N, 1)^T$ is the vector describing the instance of a statement in loop iteration (x, y). By mapping this value to the logical timestamp

$$(x, y, 0)^T$$

for all instances of statement $S1$ and to the timestamp

$$(x, y, 1)^T$$

for all instances of statement $S2$ we obtain the desired lexicographical order. Thus, we have to find matrices $\mathcal{T}_{S1}$ and $\mathcal{T}_{S2}$ such that

$$\mathcal{T}_{S1}\begin{pmatrix} x \\ y \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \qquad \text{and} \qquad \mathcal{T}_{S2}\begin{pmatrix} x \\ y \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

hold. Hence, the *schedule function* for statement $S1$ is given by the matrix

$$\mathcal{T}_{S1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{Z}^{k \times n}$$

and for statement $S2$ by the matrix

$$\mathcal{T}_{S2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \in \mathbb{Z}^{k \times n}$$

satisfying the constraints defined above. However, alternative formats of logical timestamps yield different matrices. The important thing is that those are indeed integer matrices and the resulting logical timestamps accurately describe the execution order of the instances of the involved statements.

Access Functions The third component for modeling statements in the polyhedral model are lists of access functions. Access functions map the instance of a statement to the accessed elements of an array. Thereby, read and write accesses are distinguished.

As for the schedule function, access functions are specified by an integer matrix $\mathcal{A}_{(s,m,d)} \in \mathbb{Z}^{k \times n}$ where s is the associated statement, $m \in$

$\{\text{USE}, \text{DEF}\}$ is the access mode, d identifies the accessed array, k corresponds to the number of dimensions of d and n to the length of the iteration vector.

In our example the statement $S1$ is associated with the access function

$$\mathcal{A}_{(\text{S1,USE},a)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Hence, an instance of $S1$ in iteration (x, y), addressed by the vector $\vec{i} = (x, y, N, 1)^T$ is reading the element

$$\mathcal{A}_{(\text{S1,USE},a)}\vec{i} = (x, y-1)^T$$

of array a. Similarly, $S2$ is associated with the access function

$$\mathcal{A}_{(\text{S2,DEF},a)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

describing the write access on element $a[x][y]$ in iteration (x, y).

Full Representation The full polyhedral representation of the given code fragment is given by

$$\left(\vec{I}, \left\{ \left(\mathcal{D}_{S1}, \mathcal{T}_{S1}, \left\{\mathcal{A}_{(\text{S1,USE},a)}\right\}\right), \left(\mathcal{D}_{S2}, \mathcal{T}_{S2}, \left\{\mathcal{A}_{(\text{S2,DEF},a)}\right\}\right) \right\}\right)$$

consisting of the global *iteration vector* $\vec{I}$ and a set of two tuples containing the iteration domains, affine schedule and access functions describing the two statements $S1$ and $S2$.

Polyhedral Model based Dependency Analysis

Based on the polyhedral representation a dependency test for the given loop nest can be conducted as follows [14]. To constitute a *true dependency* a write operation has to be conducted on some array element before a read operation is targeting the same element. In our example only one read instruction and one write instruction is present. Hence, only this combination has to be tested.

Lets assume instance $\vec{r} = (i_r, j_r, N, 1)^T$ is the instance of statement $S1$ reading the element written by instance $\vec{w} = (i_w, j_w, N, 1)^T$ of statement $S2$. To constitute a dependency, both instances have to exist. Hence, $\vec{r} \in \mathcal{D}_{S1}$ and $\vec{w} \in \mathcal{D}_{S2}$ which is checked by the constraint

$$\mathcal{D}_{S1}\vec{r} \geq \vec{0}$$

and

$$\mathcal{D}_{S2}\vec{w} \geq \vec{0}$$

Also they have to access the same array element, hence

$$\mathcal{A}_{(\text{S1,USE},a)}\vec{r} = \mathcal{A}_{(\text{S2,DEF},a)}\vec{w}$$

has to be satisfied which is equivalent to

$$\begin{aligned} \mathcal{A}_{(\text{S1,USE},a)}\vec{r} - \mathcal{A}_{(\text{S2,DEF},a)}\vec{w} &\geq \vec{0} \\ -\mathcal{A}_{(\text{S1,USE},a)}\vec{r} + \mathcal{A}_{(\text{S2,DEF},a)}\vec{w} &\geq \vec{0} \end{aligned}$$

Also, the read operation has to happen after the write operation. Thus

$$\mathcal{T}_{S1}\vec{r} \succ \mathcal{T}_{S2}\vec{w}$$

has to be satisfied, where $\succ$ compares the two logic timestamps lexicographically. This is the case if up to a given level l it holds that

$$(\mathcal{T}_{S1}\vec{r})_{1:l-1} = (\mathcal{T}_{S2}\vec{w})_{1:l-1}$$

and for the level l

$$(\mathcal{T}_{S1}\vec{r})_l > (\mathcal{T}_{S2}\vec{w})_l$$

where e.g. $(\mathcal{T}_{S1}\vec{r})_{1:l-1}$ denotes the first $l-1$ elements of the vector $\mathcal{T}_{S1}\vec{r}$ and $(\mathcal{T}_{S1}\vec{r})_l$ its l-th element. Those constraints can be written using linear inequalities of the shape

$$A\vec{x} + \vec{b} \geq 0$$

using

$$\begin{bmatrix} \mathcal{T}_{S1,[1-(l-1):]} & -\mathcal{T}_{S2,[1-(l-1):]} \\ -\mathcal{T}_{S1,[1-(l-1):]} & \mathcal{T}_{S2,[1-(l-1):]} \\ \mathcal{T}_{S1,[l:]} & -\mathcal{T}_{S2,[l:]} \end{bmatrix} \begin{pmatrix} \vec{r} \\ \vec{w} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ -1 \end{pmatrix} \geq \vec{0}$$

where e.g. $\mathcal{T}_{S1,[1-(l-1):]}$ denotes the first $l-1$ rows of matrix $\mathcal{T}_{S1}$. Let

$$P_{S1,l} = \begin{bmatrix} \mathcal{T}_{S1,[1-(l-1):]} \\ -\mathcal{T}_{S1,[1-(l-1):]} \\ \mathcal{T}_{S1,[l:]} \end{bmatrix}$$

and

$$P_{S2,l} = \begin{bmatrix} -\mathcal{T}_{S2,[1-(l-1):]} \\ \mathcal{T}_{S2,[1-(l-1):]} \\ -\mathcal{T}_{S2,[l:]} \end{bmatrix}$$

Then all those constraints can be combined into the inequation

$$\begin{bmatrix} \mathcal{D}_{S1} & 0 \\ 0 & \mathcal{D}_{S2} \\ \mathcal{A}_{(\text{S1,USE},a)} & -\mathcal{A}_{(\text{S2,DEF},a)} \\ -\mathcal{A}_{(\text{S1,USE},a)} & \mathcal{A}_{(\text{S2,DEF},a)} \\ P_{S1,l} & P_{S2,l} \end{bmatrix} \begin{pmatrix} \vec{r} \\ \vec{w} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ -1 \end{pmatrix} \geq \vec{0}$$

which specifies a convex polytope summarizing accurately all dependencies on a per-instance level between the read operation in $S1$ and the write operation in $S2$ for level l ($\sim$ loop level). A dependency exists iff the polytope is not empty.

For example, let $l = 2$. Then the dependency polytope is given by

$$\left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 \\ \hline -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \end{array}\right] \begin{pmatrix} i_r \\ j_r \\ N \\ 1 \\ i_w \\ j_w \\ N \\ 1 \end{pmatrix} + \begin{pmatrix} \vec{0} \\ -1 \end{pmatrix} \geq \vec{0}$$

which corresponds to the constraints

$$\begin{aligned} i_r &\geq 0 & i_r - i_w &\geq 0 \\ -i_r + N - 1 &\geq 0 & j_r - 1 - j_w &\geq 0 \\ -i_r + j_r &\geq 0 & & \\ -j_r + N - 1 &\geq 0 & -i_r + i_w &\geq 0 \\ & & -j_r + 1 + j_w &\geq 0 \\ i_w &\geq 0 & & \\ -i_w + N - 1 &\geq 0 & i_r - i_w &\geq 0 \\ -i_w + j_w &\geq 0 & -i_r + i_w &\geq 0 \\ -j_w + N - 1 &\geq 0 & j_r - j_w - 1 &\geq 0 \end{aligned}$$

which can be rewritten and reduced to

$$\begin{array}{ccc} 0 \leq i_r < N & 0 \leq i_w < N & i_r = i_w \\ i_r \leq j_r < N & i_w \leq j_w < N & j_r = j_w + 1 \end{array}$$

The first four constraints limit the ranges of the parameters of the read and write instances while the last two fix their relation. Clearly, for any $N > 1$ the corresponding polytope is not empty.

For a full coverage of the read-after-write dependencies between $S1$ and $S2$ all levels $1 \leq l < k$ where k is the length of the logical time stamp have to be considered independently. Also, other kind of dependencies need to be considered as well.

Limitations

A polyhedral representation of a code fragment can only be obtained for cases where the control flow is determined by affine constraints on loop iterators or global (constant) parameters. Hence, e.g. all for-loop boundaries and conditionals in the targeted code are limited by this restriction. A code fragment fulfilling this criteria is referred to as a *Static Control Part* (SCoP). Constructs violating this restriction may be over-approximated, resulting in reduced accuracy. Also, to accurately model array accesses, the corresponding subscripts are equally restricted to affine expressions or required to be over-approximated. Nevertheless, it has been found that a large portion of computation time, in particular in scientific applications, is spent in SCoPs [15]. Together with its precision, this circumstance lays the foundation for the great success of the polyhedral model in the field of program analysis and optimizations.

4.5.2 Integration of Polyhedral Analyses

The only component required to support polyhedral model based analysis on top of the Insieme IR is a utility capable of extracting a polyhedral representation from a given IR code fragment. Thereby, compared to low-level IRs, the high level nature of INSPIRE eliminates the requirement of identifying loops, arrays, boundary and subscript expressions. Also, compared to C/C++ input code, the fact that *for*-loops are always representing count-controlled loops simplifies the task. The coverage of the polyhedral model can be further extended by (implicitly) inlining statically bound function calls, constant propagation and the symbolic evaluation of arithmetic values provided by the corresponding constraint-based analysis.

To conduct analyses, the extracted polyhedral representation, referencing statements in the IR DAG, is exported into a format accepted by some third-party library, specifically the integer set library *isl* [105]. Based on this, arbitrary PM based analyses may be realized.

4.6 Dynamic Analyses

Beside static analyses, which try to deduce properties of a program from its representation, dynamic analyses target features observed during the execution of a program itself. In particular, those include non-functional properties like the execution time or power consumption of a program. In the Insieme infrastructure dynamic analyses are supported utilizing a combination of compiler and runtime features.

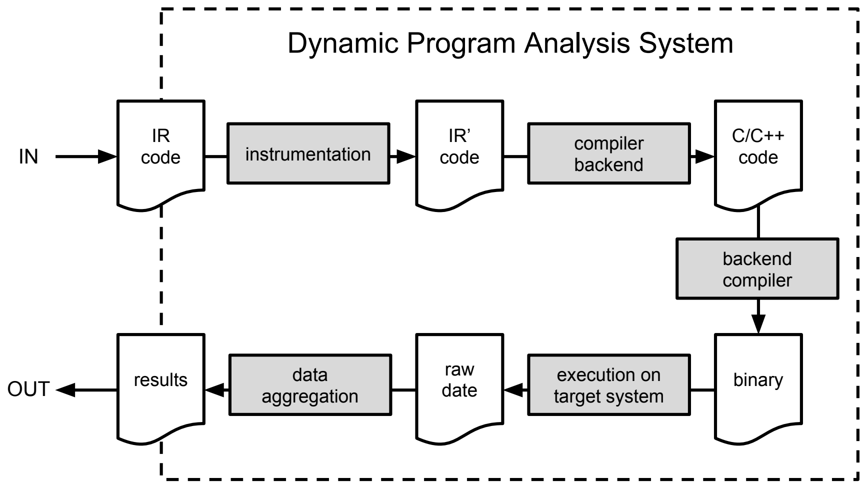


Figure 4.2: Overview on a system for dynamic program analyses.

4.6.1 Overview on Dynamic Analyses

Figure 4.2 illustrates the basic architecture of a dynamic program analysis system. A program within the compiler, represented in its IR form, is passed to the dynamic analysis system together with a list of metrics to be obtained (not shown in the picture). Within the system, an instrumented version of the program is created, converted into target code, compiled by a backend compiler, e.g. GCC or icc, and executed on the target system. The instrumentation code, combined with corresponding runtime support is collecting information regarding the execution which is further processed and aggregated and returned to the user of the dynamic analysis.

To realize dynamic analyses, support of the runtime system to conduct the actual measurements is required. Therefore, an interface enabling the compiler to address code regions and metrics of interest is needed as well as a specification of the data format utilized to return obtained results.

Depending on the runtime support, dynamic analyses may cover various aspects of a program execution. Those may include

- execution times
- power consumption
- maximum / average energy dissipation
- parallel efficiency and load balancing information
- data dependencies
- statistics on the actual control flow of a code fragment

- statistics on data transfers
- hardware or software stack events
- values expressions within the code fragment are evaluated to

Some of the data to be obtained may be simply counted, others measured or derived through a model based approach.

Limitation

The major limitation of dynamic analyses is based on the fact that actual input is required for analyzing a given code fragment. Depending on the application and the observed metric, the selection of the input may have a significant impact on the obtained results. Also, several repetitions of measurements may have to be conducted to obtain significant results.

4.6.2 Integration of Dynamic Analyses

Due to the architecture of the Insieme infrastructure support for dynamic analysis can be realized by utilizing the various components of the Insieme Runtime System, in particular the monitoring and event system (see page 34). An example interface for obtaining non-functional measurements, including time, power, energy and parallel efficiency metrics, is given by a function

$$measure : (2^{\mathcal{A}}, 2^{\mathcal{M}}, \mathbb{N}, \mathcal{C}) \rightarrow (2^{\mathcal{A}} \rightharpoonup (2^{\mathcal{M}} \rightharpoonup \mathcal{Q}^*))^*$$

where $\mathcal{A}$ is the set of node addresses, $\mathcal{M}$ is the set of metrics, $\mathcal{C}$ is an (optional) configuration determining options for the backend compiler, the runtime system and the selection of the target machine, and Q a set of quantities, hence the results to be obtained. Let $a_1, a_2 \in \mathcal{A}$ be node addresses targeting statement or expressions within an IR fragment, $m_1, m_2, m_3 \in \mathcal{M}$ be metrics, $n \in \mathbb{N}$ the number of desired repetitions and $c \in \mathcal{C}$ a configuration for the backend compiler, the runtime system and the target machine. Then a call

$$\text{measure}(\{a_1, a_2\}, \{m_1, m_2, m_3\}, n, c)$$

yields a list of n mappings – one entry for each of the execution – where each mapping assigns to each combination of an address and metric a list of quantities enumerating the corresponding measurement results collected during the corresponding execution. For instance, let res be obtained by the given function call, a_1 be a reference to the body of a loop and m_1 a metric representing the execution time. Then the value

$$res_2[a_1][m_1] \in \mathcal{Q}^*$$

is a list of the execution times of each iteration of the corresponding loop encountered during the second execution of the code fragment.

The necessary instrumentation, compilation, execution and data aggregation steps are covered generically by the implementation of the dynamic analysis system. At its current development state, support for the measurement of execution time, energy, power and parallel efficiency in addition to a plethora of hardware counters is offered. Furthermore, derived metrics, in particular including ratios and aggregates of other metrics, may be defined by composing existing metrics. The resulting quantities $\mathcal{Q}$ are equipped with units, e.g. joule or nano-seconds, such that those may be safely utilized within arithmetic operations.

Among the options offered for the backend compiler, the runtime system and the target machine, are compiler flags, runtime thread management and scheduling policies, environment variable options and the possibility of providing an executor handling the actual program execution. In the default setup processes are executed locally, on the same machine the Insieme compiler instance is running on, while alternative implementations provide the possibility of processing codes on remote machines utilizing ssh sessions or even job submission systems as they are encountered on clusters.

With its ability to obtain non-functional parameters of application codes, the dynamic program analysis system within the Insieme Compiler constitutes one of the foundations of iterative optimization approaches where input codes are gradually adjusted and evaluated during the course of their compilation.

4.7 Summary

In this chapter techniques for analyzing programs encoded utilizing the Insieme IR have been presented. It started by introducing rudimentary techniques, covering e.g. feature extraction facilities, in Section 4.3.2. Those facilities have, for instance, been utilized by Insieme based utilities for characterizing codes [56].

The chapter continued with more sophisticated techniques. A flow-, thread- and call-context-sensitive program analysis framework based on a constraint based analysis approach [68] has been developed and presented by Section 4.4. Its central component, an advanced lazy constraint solver algorithm supporting local restarts for increased performance, flexibility and precision, has been developed as part of this thesis (Section 4.4.3). Unlike conventional CBA frameworks operating on sequential code, the framework has been adapted to consider the parallel nature of the processed program description. Existing related work targeting parallel codes is based on the less flexible, yet conventional, data flow analysis approach and hence depending on a static knowledge about the parallel structure of the targeted code [108, 40] or its extraction using preprocessing steps [51, 115]. Unlike those, the framework developed in this chapter is deducing this information from the input program directly utilizing a combination of analyses contributing constraints to a single analysis problem instance (see Section 4.4). This can lead to precision increasing, mutual interactions between e.g. data- and (parallel) control-flow analyses. Also, exceeding the capabilities of related work [34, 46], the synchronizing and ordering effects of channel communication are properly respected (see Example 4.24). Furthermore, generic utilities for modeling composed values – as they are encountered in high-level IRs – have been developed in Section 4.4.4.

Section 4.5 outlined the integration of polyhedral model based analyses into the Insieme compiler infrastructure. The high-level nature of our IR, in particular the preservation of information regarding loop and array constructs, eases the integration of PM based techniques. Among others, this techniques have been utilized within Insieme based applications for determining loop dependencies to check the validity of transformations, for instance for the application covered in Section 6.2, as well as to estimate the computational complexity of loop nests as described in Section 6.3.2.

Finally, Section 4.6 covered means developed for the Insieme infrastructure to utilize the monitoring capabilities of the runtime system to obtain non-functional properties from a code fragment by observing its execution. This support for dynamic analyses has, for instance, been utilized for realizing the iterative compilation schema outlined in Section 6.2.2.

Chapter 5

Transformations

After introducing the Insieme IR in Chapter 3 and developing a tool set for analyzing it in Chapter 4, the final step in establishing an infrastructure for optimizing parallel programs is to provide utilities for actually manipulating processed codes. This is the topic of this chapter.

Section 5.2 starts by covering the basic mechanisms and a set of rudimentary utilities for transforming IR codes as well as the interfaces allowing arbitrary transformations to be implemented. It is followed by a description of a higher-level, pattern based rewriting system developed for the Insieme Compiler to abstract the specification of code transformations in Section 5.3. After this, means to incorporate transformations utilizing the powerful capabilities of the polyhedral model are covered in Section 5.4. Finally, Section 5.5 covers utilities to compose individual transformation steps implemented utilizing arbitrary techniques to establish transformation scripts to be applied on IR codes. The result is an infrastructure providing an extensible list of parametrized transformations that can be flexibly composed to conduct arbitrary manipulations on programs encoded utilizing the Insieme intermediate representation.

5.1 Contributions

The major contributions of this chapter are:

- a rudimentary framework for conducting arbitrary code manipulations on the Insieme compiler IR preserving invariants (Section 5.2)
- the development of a novel, pattern matching based transformation infrastructure utilizing the high-level, term like structure of the Insieme IR for specifying code manipulations (Section 5.3)
- the demonstration of the integration of polyhedral model based loop transformations into the Insieme compiler infrastructure (Section 5.4)

- the development of a framework for the structured composition and orchestration of code manipulations (Section 5.5)

In the context of this thesis, this chapter demonstrates that the increased complexity of a high-level compiler IR, compared to the simple structure of a low-level IR, can still be adequately handled when conducting code manipulations. Even more, the presents of high-level information in the IR enables the specification of code transformation on a higher level of abstraction, and hence in a more natural and/or efficient way.

5.2 Transforming the IR

One of the design concepts of the data structure constituting the Insieme IR is the sharing of nodes and, as a consequence, the immutability of IR structures as outlined in Section 3.10. Thus, an IR code fragment can not be transformed by navigating to a node utilizing the IR navigation utilities and altering member fields of the node. Instead, a modified copy has to be created. This, most basic transformation step, is supported by *node mappers* as they are covered in the following sub-sections as well as a set of derived utilities. Those utilities provide the foundation for building operations conducting arbitrary code manipulations on the Insieme IR.

5.2.1 Node Mappers

The foundation of all program analyses has been laid by *visitors* – one central, generic piece of infrastructure to navigate the IR. The counterpart for transforming IR constructs is provided by the *node mapper* infrastructure.

However, first, as a foundation of all transformations within this chapter, a formal definition of the transformed data structure is required.

Definition 5.1 (IR structure)**.** The set $\mathbb{T} \cup \mathbb{E} \cup \mathbb{S}$ being the union of all IR types $\mathbb{T}$, expressions $\mathbb{E}$ and statements $\mathbb{S}$, hence all IR structures, is denoted by $\mathbb{IR}$. Furthermore, let the pair

$$n = (k, c) = (k, [c_1, \ldots, c_n]) \in \mathcal{K} \times \mathbb{IR}^*$$

denote a node $n \in \mathbb{IR}$ where $k \in \mathcal{K}$ with

$$\mathcal{K} = \{\text{abstract type}, \text{struct type}, \text{variable}, \text{lambda}, \text{if}, \ldots\}$$

denotes the *kind* of the node and $c \in \mathbb{IR}^*$ the list of *child nodes*.

Note that this definition implicitly defines the (logical) tree shape of IR structures. It further provides formal means for inspecting and constructing IR structures in the following definitions.

A very direct approach to implement code transformations on the immutable Insieme IR would be navigate through the IR structures and to simply create modified versions of the nodes to be altered. However, this requires to consider and respect the restrictions on the structure of all the individual node kinds – a process that is generalized by the *node mapper infrastructure.*

The node mapper infrastructure, forming the foundation of all structured IR transformations, is constituted by two elements – an abstract definition of a *node mapper* that can be freely concretized to implement transformation operations and a *map* function conducting the actual manipulations based on a concrete definition of a mapper.

Definition 5.2 (node mapper)**.** A *node mapper* is a function

$$m : \mathbb{IR}^* \to \mathbb{IR}^*$$

mapping a sequence of IR structures to a sequence of (different) IR structures. The set of all node mappers is denoted by $\mathcal{M}$.

Based on the definition of a node mapper, the *map* function conducting the actual transformation can be defined as follows.

Definition 5.3 (node transformation)**.** The function $\text{map} : (\mathbb{IR} \times \mathcal{M}) \to \mathbb{IR}$ defined by

$$\text{map}(n, m) = \text{map}((k, c), m) = \begin{cases} (k, m(c)) & \text{if } (k, m(c)) \in \mathbb{IR} \\ \text{undefined} & \text{otherwise} \end{cases}$$

obtains for a given node $n = (k, c) \in \mathbb{IR}$ a transformed version $n' = (k, m(c)) \in \mathbb{IR}$ in case the modified child list $m(c) \in \mathbb{IR}^*$ is a valid child list for the node kind $k \in \mathcal{K}$, hence $n' \in \mathbb{IR}$.

By properly defining mapper functions, any transformation may be conducted on a given code fragment.

Example 5.1 (node mapper)**.** Let $s =$ 'if (c_1) then s_1 else s_2' $\in \mathbb{S} \subset \mathbb{IR}$ be a statement represented by the IR node

$$(\text{if}, [c_1, s_1, s_2]) \in \mathbb{IR}$$

To transform s into $s' =$ 'if (c_2) then s_2 else s_1' a mapper $m \in \mathcal{M}$ such that

$$m([c_1, s_2, s_1]) = [c_2, s_2, s_1]$$

is created such that

$$\begin{aligned}
\text{map}(s, m) &= \text{map}(\text{'if } (c_1) \text{ then } s_1 \text{ else } s_2\text{'}, m) \\
&= \text{map}((\text{if}, [c_1, s_1, s_2]), m) \\
&= (\text{if}, m([c_1, s_1, s_2])) \\
&= (\text{if}, [c_2, s_2, s_1]) \\
&= \text{'if } (c_2) \text{ then } s_2 \text{ else } s_1\text{'} \\
&= s'
\end{aligned}$$

assuming $s' \in \mathbb{IR}$.

The true capabilities of the node mapper infrastructure are only revealed when utilizing recursive node mapper definitions.

Example 5.2 (recursive node mapper)**.** Consider the problem of defining a function

$$\text{substitute} : (\mathbb{IR} \times \mathbb{IR} \times \mathbb{IR}) \to \mathbb{IR}$$

such that substitute(x, y, z) transforms $x \in \mathbb{IR}$ by replacing all occurrences of $y \in \mathbb{IR}$ by $z \in \mathbb{IR}$. Such a function can be defined utilizing the map function by

$$\text{substitute}(n, a, b) = \text{map}(n, m_{ab})$$

where $m_{ab} \in \mathcal{M}$ is a *node mapper* defined by

$$m_{ab}([c_1, \ldots, c_n]) = [s_{ab}(c_1), \ldots, s_{ab}(c_n)]$$

based on the function $s_{ab} : \mathbb{IR} \to \mathbb{IR}$ defined by

$$s_{ab}(n) = \begin{cases} b & \text{if } n = a \\ \text{substitute}(n, a, b) & \text{otherwise} \end{cases}$$

In this (mutual) recursive definition the *node mapper* utilizes the function to be defined to conduct modifications inductively over the recursive definition of the underlying IR structure.

5.2.2 Manipulation Toolbox

While providing a universal utility for transforming IR structures, implementing *node mappers* for each individual manipulation step can be a cumbersome task – in particular since similar operations are frequently required.

Thus, to simplify manipulating operations on the IR, a variety of higher level transformation primitives have been implemented, utilizing the node mapper as their foundation. Those include:

- *substitution operations* – replacing all occurrences of a given node by another in a given code fragment, a list of nodes by a list of other nodes, an individual node addressed by a node address by another node, or a list of individual nodes addressed by node addresses by other nodes

- *variable handling* – replacing a variable by another variable or expression of the same type or replacing a variable by a variable or expression of another type; In the latter case, occurring typing issues, like e.g. the altered type of a function in case the modified variable has been one of the parameters, are automatically corrected – if possible. Also, those operations may be limited to the scope of a local function or recursively decent into any reachable function.

- *statement handling* – operations to insert, remove or move statements within a given code fragment

- *function handling* – operations to inline function calls to expressions (for simple functions bodies) or statements (more complex function bodies) and the reverse operation outlining expressions or statements to function calls – optionally by capturing data of the current scope utilizing a bind expression to obtain a desired signature for the resulting function. The later is, for instance, utilized to created thread bodies or pfor bodies when transforming sequential code into parallel code.

- *recursive function and type handling* – operations for unfolding and unrolling recursive function and type definitions. Those operations are the recursive equivalents to loop-peeling and loop-unrolling. For instance, consider the recursive type

  $$rec\ \alpha.\{\alpha = struct\{v : bool; n : ref\ \langle rec\ \alpha\rangle\}\}$$

  An unfolding operation yields the type

  $$struct\{v : bool; n : ref\ \langle rec\ \alpha.\{\alpha = struct\{v : bool; n : ref\ \langle rec\ \alpha\rangle\}\}\rangle\}$$

  while an unrolling operation yields the type

  $$rec\ \alpha.\{\alpha = struct\{v : bool; n : ref\ \langle struct\{v : bool; n : ref\ \langle rec\ \alpha\rangle\}\}\rangle\}$$

  Those operations also handle mutual recursive types and functions. While the unfolding operations are utilized e.g. during the code generation in the backend or type checks to handle recursive structures, the unrolling operations are used, for instance, in combination with function inlining by code optimizations targeting the reduction of function call overheads in recursive codes (see Section 6.4).

- *instantiation of generic types and functions* – creating versions of types and functions by substituting type variables by (concrete) types. Required, e.g. in the backend, to create code for generic structures.
- *data propagation* – a transformation capable of providing access to data of a surrounding scope in a nested function body by forwarding a reference or copy as an additional parameter through all intermediate function calls; This way information can be made available within a nested function without the unfavorable utilization of global variables.

Each of those transformation primitives is implemented as a single function and can thus be flexibly combined with other manipulation steps to realize desired actions on an IR code fragment.

Furthermore, the implementation of those primitives is tuned for efficiency utilizing in particular memoization whenever possible. Also, steps descending into sub-trees, for which it can safely be assumed that they are unaffected by a given manipulation, are skipped. Finally, utilities ensuring valid code fragments are employed implicitly, freeing the user of those primitives from taking care of several side-effects of their transformations. For instance, given the code fragment

```
let f = (struct { a : int<4> } x) ⇒ int<4> {
  return x.a;
};

struct { a : int<4> } y;
f(y);
```

If the type of y is altered to

```
struct { a : uint<8> }
```

and the proper transformation primitive is utilized, then this step also updates the parameter type of the function f and its return type accordingly, such that the resulting code fragment corresponds to

```
let f = (struct { a : uint<8> } x) ⇒ uint<8> {
  return x.a;
};

struct { a : uint<8> } y;
f(y);
```

Those implicit aids in code transformation utilities enable the efficient development of more complex code manipulation operations.

5.2.3 Handling Annotations

As introduced in Section 3.10.1, the data structures utilized to represent the Insieme IR are exclusively covering the IR constructs. However, additional, generic information may be annotated to each node instance. Those annotations may, for instance, contain cached analysis results. Unlike the IR structures itself, those annotations are mutable and may hence be freely modified. Thus, to alter annotations it is sufficient to obtain a reference to them, e.g. by utilizing a *visitor* or some derived utility to obtain the node the targeted annotation is attached to. Once access to a node has been obtained, annotations may be freely added, updated or removed – without the requirement of altering the IR structure.

One problem of annotations is that they are bound to a single node instance. If a modified version of this node is created, annotations are not automatically migrated and the contained information would be lost for the new version. For instance, given a node $n \in \mathbb{IR}$ with an annotation a. If n is transformed into $n' \in \mathbb{IR}$, hence n' is a modified copy of n, then the annotation a will still remain with n and not with n'. However, sometimes the conducted operation may not have influenced the information stored in a, thus having a also attached to n' would be desirable.

In general, due to the versatile nature of annotations, no universal regulation on when annotations should be preserved upon modifications can be established. Hence, this decision is left to the annotations themselves.

Definition 5.4 (annotation migration)**.** Let $\mathcal{A}$ be the set of annotations. For each annotation $a \in \mathcal{A}$ there is a function

$$m_a : (\mathbb{IR} \times \mathbb{IR}) \to 2^{\mathcal{A}}$$

that determines for a node $n \in \mathbb{IR}$ and its transformed copy $n' \in \mathbb{IR}$ the set of annotations $m_a(n, n')$ to be attached to node n' if $a \in \mathcal{A}$ was attached to n. Let $A : \mathbb{IR} \rightharpoonup 2^{\mathcal{A}}$ a partial mapping associating IR nodes with their attached annotations. Then the function

$$\text{migrate} : (\mathbb{IR} \times \mathbb{IR} \times (\mathbb{IR} \rightharpoonup 2^{\mathcal{A}})) \to (\mathbb{IR} \rightharpoonup 2^{\mathcal{A}})$$

defined by

$$\text{migrate}(n, n', A) = A\left[n' \mapsto A[n'] \cup \bigcup_{a \in A[n]} m_a(n, n')\right]$$

obtains the updated annotation assignment after transforming n into n'.

By individually defining m_a each annotation a can independently decide on whether to be migrated to a transformed version of a node it is attached to or not. It may also decide to alter the contained information and to attach a transformed version of itself to the resulting node. Even multiple annotations may be attached.

Example 5.3 (annotation migrations)**.** If an annotation $a \in \mathcal{A}$ desires to be preserved upon any manipulation it defines its migration function m_a by

$$m_a(n, n') = \{a\}$$

In case the annotation shall not be preserved upon manipulations m_a can be defined by

$$m_a(n, n') = \emptyset$$

and in case a modified version $f(n', a) \in \mathcal{A}$ shall be attached, the definition could be equal to

$$m_a(n, n') = \{f(n', a)\}$$

which will result in the desired behavior.

The migrate function of Definition 5.4 may be explicitly called after constructing a modified version of a given node. It is also implicitly invoked by all transformations conducted by the primitives outlined above by its integration into the node transformation function *map*.

Definition 5.5 (annotated node transformation)**.** Let $\mathbb{IR}_a = \mathbb{IR} \times (\mathbb{IR} \rightharpoonup 2^{\mathcal{A}})$ denote the set of annotated IR structures where each element $(i, A) \in \mathbb{IR}_a$ consists of an IR structure $i \in \mathbb{IR}$ and an annotation mapping $A \in \mathbb{IR} \rightharpoonup 2^{\mathcal{A}}$. Then the function

$$\text{map} : (\mathbb{IR}_a \times \mathcal{M}) \to \mathbb{IR}_a$$

defined by

$$\text{map}((n, A), m) = (\text{map}(n, m), \text{migrate}(n, \text{map}(n, m), A))$$

extends the function map of Definition 5.3 by annotation migration capabilities.

This concludes the infrastructure available for handling IR node annotations. For simplicity, annotations and their handling within transformations are omitted throughout the rest of this chapter. However, the utilization of the available utilities can be assumed.

5.3 Pattern Based Transformations

Even based on the available manipulation primitives, code transformations are still implemented by a (long) hand-coded sequence of operations inspecting and manipulating the utilized internal IR – a situation to be faced by any high-level source-to-source compiler infrastructure. This approach is not only labor-intensive and error-prone, thus limiting productivity, but even more crucially, it also reduces maintainability due to its tendency to result in obscure code [106]. A more structured and concise approach is desired.

Although rarely encountered within widely utilized compiler infrastructures, transformation systems tackle this issue by offering a declarative, rule based interface for the definition of transformations [107, 24, 4]. Fundamentally, each rule consists of a pattern and a replacement template. Any term matching the pattern is replaced by an instantiation of the template. In general, the pattern matching is based on unification, resulting in two restrictions. On the one hand, unification can only impose limited constraints on the matched terms. For instance, it can not check whether within an arbitrary nesting level of a term a given sub-term occurs. On the other hand, unification builds upon an algebraic structure, hence terms of the targeted structure are restricted to a fixed arity [107, 24]. Yet, within AST-like IRs like our own, constructs which do not naturally exhibit a fixed arity (e.g. compound statements or argument lists) are omnipresent. Frequently this issue is circumvented by representing variable-sized lists using artificial function symbols (or grammar rules) linking individual elements to recursively composed lists. However, not only does this approach conceal the structure of the internal representation, it also increases the complexity of defining patterns and rewrite rules.

Within this section we present a novel combination of unification-based term rewriting rules and regular expressions allowing the concise declarative description of complex transformations for arbitrary tree structures – in particular high-level compiler IRs like our own. The content of this section has been published [50].

5.3.1 Design Goals

This section provides an informal overview of the intended capabilities of our pattern matching and rewriting system by discussing a list of example use cases. A simple case, which is already not (directly) supported by unification based approaches, is to check whether a given variable $v1$ is present within some code fragment. We would like to write a pattern similar to

$$aT(v1) \tag{5.1}$$

where the construct aT denotes *"anywhere in the tree"*. A more extended case would be the requirement to check whether the expression $exp1$ is present as a full expression within a given compound statement. In this case we would like to write something similar to

$$\{\_^*, exp1, \_^*\} \tag{5.2}$$

where $\_$ denotes a wildcard, * the Kleene operator (any number of repetitions, including none) and the brackets $\{\}$ the enclosing compound statement.

In many cases patterns will be defined not only to constrain the structure of some term but also to extract information. For instance, it might be necessary to obtain the variable being declared by some declaration statement. In this case we would like to use a pattern involving a variable $\$x$ similar to

$$decl(\$x) \tag{5.3}$$

to obtain the requested information. Matched against the input declaration

$$int\ a\ =\ 5 \tag{5.4}$$

pattern (5.3) should yield the variable mapping $[x \mapsto a]$. Furthermore, in a case where all variables declared within a compound statement should be obtained, the pattern

$$\{(\neg decl(\_))^*, (decl(\$x), (\neg decl(\_))^*)^*\} \tag{5.5}$$

matched against

$$\{int\ a\ =\ 5;\ f(a);\ bool\ b\ =\ true;\ int\ c\ =\ 7;\} \tag{5.6}$$

should yield $[x \mapsto [a, b, c]]$. It should also be possible to constrain the values pattern variables are bound to. For instance, if the selection should be limited to declarations of integer variables we would like to use a pattern similar to

$$\{(\neg decl(var(int, \_))^*, (decl(\$x : var(int, \_)), (\neg decl(var(int, \_))^*)^*\} \tag{5.7}$$

Here, $var(t, n)$ is a pattern construct matching an IR variable n of type t and the construct $\$x : y$ defines a pattern variable x matching every structure satisfying the pattern y. Applying this pattern to fragment (5.6) should result in $[x \mapsto [a, c]]$.

In some cases we want more. For instance, we might require a pattern identifying variables being declared but never used. This should be covered by

$$\{decl(\$x), (\neg aT(\$x))^*\} \tag{5.8}$$

Hence, a declaration of some IR variable captured by the pattern variable $\$x$ followed by a sequence of statements not including this particular variable. Note the difference to the previous examples. In pattern (5.5) the pattern variable $\$x$ is bound to a list of sub-trees, once for each declaration in the list, while in the current example $\$x$ should be bound only once to the variable being declared at the beginning of the compound statement. In the subsequent repetition $(\neg aT(\$x))^*$ the variable is supposed to be the fixed to the value previously bound to $\$x$.

From this observation we derived the following desirable rule for variable bindings: within every iteration of a repeating sub-pattern p, variables may be re-bound to new values, unless already bound before entering p the first time. If they were already bound, then, within all repetitions, previously bound variables remain bound to the value determined before entering p.

Of course, pattern (5.8) is limited to situations where the unused variable is declared by the first statement in a compound. This restriction is lifted by using

$$aT(decl(\$x)) \land \{(\neg aT(\neg decl(\$x) \land \_(\_^*, \$x, \_^*)))^*\} \tag{5.9}$$

where $\land$ is the conjunction of patterns and $\_(\_^*, p, \_^*)$ matches any node with a child matching the pattern p. The pattern searches for any sub-tree declaring a variable $\$x$ which is never referenced outside the declaration.

As a final challenge for the pattern syntax we would like to define a pattern capable of listing all for loops within a perfectly nested loop nest. The problem here is that the loop nest might be arbitrarily deep. The Kleene operator is limited to horizontal matching, along the list of children of a single node. For this class of use cases an operator defining a recursively nested tree structure is required. We define an additional primitive $rT.x(p)$ which is equivalent to p with the additional effect of binding the pattern p to the recursive variable x. Within p the term $rec.x$ can be used as a placeholder for the full pattern p. Based on this operators we can define a pattern for a for-loop nest using

$$rT.x(\$l : forStmt(\neg forStmt(\_) \lor rec.x)) \tag{5.10}$$

where $forStmt(b)$ matches any for-loop with a body matching the pattern b. The variable $\$l$ will be bound to a list of all loops of the matched loop nest.

Finally, patterns should provide means to match the input for transformation rules. For instance, the rule

$$\{\$xs, \{\}, \$ys\} \rightarrow \{\$xs, \$ys\} \tag{5.11}$$

is designed to match any compound statement which includes an empty compound statement and to eliminate this inner statement. On the right hand side of the rule the replacement is specified by a template utilizing the variables of the left hand side. Also, unlike all previous examples, in this case the pattern variables $\$xs$ and $\$ys$ match lists of trees instead of individual trees.

5.3.2 Patterns and Replacements

Based on this desired properties of the IR tree pattern matcher and rewriting system the necessary formalism satisfying those requirements is developed within this section.

The approach is divided into two layers – the *core primitives*, defining its expressive power and a variety of *derived constructs* created by composing core primitives to provide more user-friendly, domain-specific connectors. This separation enables essential algorithms including the pattern matching to focus on a minimal set of constructs while the developers utilizing the system can define patterns and rules using constructs customized for their specific domain, in particular our IR.

Within this sections the formal foundation and the core primitives of our patterns and replacements are specified while derived constructs are covered within the following two sections.

Tree Structure

Before defining a grammar for patterns and replacements a definition of the structures to be operated on has to be provided.

Definition 5.6 (universal tree structure)**.** Let $\mathbb{A}$ be a set of atomic values (e.g. the union of integers, names and booleans) and $\mathbb{K}$ be a set of the node types to be distinguished. Any tree generated by the production

$$T ::= a \mid k(T^*)$$

where $a \in \mathbb{A}$ and $k \in \mathbb{K}$ is a valid input for our infrastructure.

This definition is generic enough to cover arbitrary trees including the structure of the Insieme IR (see Definition 5.1). In particular it does not depend on a fixed arity for any node type $k \in \mathbb{K}$. For the remainder of this section, let $\mathbb{T}$ be the set of all trees generated by the production rule given above for suitable sets $\mathbb{A}$ and $\mathbb{K}$.

Patterns

Within our framework we distinguish two kinds of patterns – tree and list patterns. While tree patterns describe the structure of individual trees, list patterns cover the composition of lists of trees (=forests).

Definition 5.7 (tree and list pattern syntax)**.** Let $\mathbb{V}$ be a set of variable identifiers. Tree patterns ϕ and list patterns ψ are generated by the following pair of mutually recursive production rules

$$\begin{aligned}
&\phi ::= \_ \mid t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \_(\psi) \mid k(\psi) \mid x : \phi \mid aT(\phi) \mid rT.x(\phi) \mid rec.x \\
&\psi ::= \epsilon \mid \phi \mid \psi, \psi \mid \psi \vee \psi \mid x : \psi \mid \psi^*
\end{aligned}$$

where $t \in \mathbb{T}$, $k \in \mathbb{K}$, $x \in \mathbb{V}$ and ϵ is the empty list. Furthermore, let Φ be the set of all possible tree patterns and Ψ be the set of all list patterns.

The semantic of patterns is defined based on the sets of trees and forests matched by those, as defined in the following section.

Semantics The semantics of patterns is specified by defining the sets $T_\phi \subseteq \mathbb{T}$ and $T_\psi \subseteq \mathbb{T}^*$, matched by a tree pattern $\phi \in \Phi$ and a list pattern $\psi \in \Psi$ respectively.

Definition 5.8 (tree and list pattern semantic)**.** Let $t, m, n, r \vdash \phi$ denote the fact that the tree $t \in \mathbb{T}$ matches the tree pattern ϕ, and $s, m, n, r \vdash \psi$ denote the fact that the sequence $s \in \mathbb{T}^*$ matches the list pattern ψ, based on the *tree variable mapping* $m : \mathbb{V} \rightharpoonup \mathbb{T}$, the *list variable mapping* $n : \mathbb{V} \rightharpoonup \mathbb{T}^*$, and the *recursive context map* $r : \mathbb{V} \rightharpoonup (\Phi \times (\mathbb{V} \rightharpoonup \mathbb{T}) \times (\mathbb{V} \rightharpoonup \mathbb{T}^*))$. The following rules provide a inductive definition of the relation $\vdash$ based on the structure of a tree pattern ϕ

$t, m, n, r \vdash \_$	iff	$true$	(wildcard)
$t, m, n, r \vdash t$	iff	$t = t$	(constant)
$t, m, n, r \vdash \neg\phi$	iff	not $t, m', n', r \vdash \phi$ and $m \sqsubseteq m'$ and $n \sqsubseteq n'$	(negation)
$t, m, n, r \vdash \phi_1 \wedge \phi_2$	iff	$t, m', n', r \vdash \phi_1$ and $t, m, n, r \vdash \phi_2$ and $m' \sqsubseteq m$ and $n' \sqsubseteq n$	(and)
$t, m, n, r \vdash \phi_1 \vee \phi_2$	iff	$t, m, n, r \vdash \phi_1$ or $t, m, n, r \vdash \phi_2$	(or)
$t, m, n, r \vdash \_(\psi)$	iff	$t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(any node)
$t, m, n, r \vdash k(\psi)$	iff	$t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(node)
$t, m, n, r \vdash x : \phi$	iff	$t, m \setminus \{x\}, n, r \vdash \phi$ and $m[x] = t$	(var)
$t, m, n, r \vdash aT(\phi)$	iff	$t, m, n, r \vdash \phi$ or $t, m, n, r \vdash \_(\_^*, aT(\phi), \_^*)$	(any tree)
$t, m, n, r \vdash rT.x(\phi)$	iff	$t, m', n', r\,[x \mapsto (\phi, m, n)] \vdash \phi$ and $m \sqsubseteq m'$ and $n \sqsubseteq n'$	(recursion)
$t, m, n, r \vdash rec.x$	iff	$r[x] = (\phi, m', n')$ and $t, m'', n'', r \vdash \phi$ and $m' \sqsubseteq m''$ and $n' \sqsubseteq n''$	(rec. end)

and a list pattern ψ

$s, m, n, r \vdash \epsilon$	iff	$s = \epsilon$	(empty)
$s, m, n, r \vdash \phi$	iff	$s = [t]$ and $t, m, n, r \vdash \phi$	(single)
$s, m, n, r \vdash \psi_1, \psi_2$	iff	$s = s_1, s_2$ and $s_1, m, n, r \vdash \psi_1$ and $s_2, m, n, r \vdash \psi_2$	(sequence)
$s, m, n, r \vdash \psi_1 \vee \psi_2$	iff	$s, m, n, r \vdash \psi_1$ or $s, m, n, r \vdash \psi_2$	(or)
$s, m, n, r \vdash x : \psi$	iff	$s, m, n \setminus \{x\}, r \vdash \psi$ and $m[x] = s$	(var)
$s, m, n, r \vdash \psi^*$	iff	$s = \epsilon$ or $s = s_1, s_2$ and $s_1, m_1, n_1, r \vdash \psi$ and $s_2, m_2, n_2, r \vdash \psi*$ and $m \sqsubseteq m_1$ and $m \sqsubseteq m_2$ and $n \sqsubseteq n_1$ and $n \sqsubseteq n_2$	(repetition)

where all free variables are existentially quantified, s_1, s_2 denotes the concatenation of two sequences s_1 and s_2, and $a \sqsubseteq b$ for two mappings a and b holds whenever $\forall x \in \mathrm{dom}(a)\ .\ a[x] = b[x]$.

A tree $t \in \mathbb{T}$ is an element of T_ϕ if there are mappings m and n such that $t, m, n, \emptyset \vdash \phi$ holds. Correspondingly, a sequence $s \in \mathbb{T}^*$ is an element of T_ψ if there are mappings m and n such that $s, m, n, \emptyset \vdash \psi$ holds.

Example 5.4 (tree and list patterns)**.** A tree

$$t = a(b, c(d, e))$$

is matched by the tree pattern $\_ \in \Phi$ since any tree is matched by the wildcard pattern. It is also matched by e.g. the pattern

$$a(b \vee c, \neg d) \in \Phi$$

since all of

$$\begin{aligned}
a(b, c(d, e)), \epsilon, \epsilon, \epsilon &\vdash a(b \vee c, \neg d) && \text{(node)}\\
[b, c(d, e)], \epsilon, \epsilon, \epsilon &\vdash b \vee c, \neg d && \text{(sequence)}\\
[b], \epsilon, \epsilon, \epsilon &\vdash b \vee c && \text{(single)}\\
b, \epsilon, \epsilon, \epsilon &\vdash b \vee c && \text{(or)}\\
b, \epsilon, \epsilon, \epsilon &\vdash b && \text{(constant)}\\
[c(d, e)], \epsilon, \epsilon, \epsilon &\vdash \neg d && \text{(single)}\\
c(d, e), \epsilon, \epsilon, \epsilon &\vdash \neg d && \text{(negation)}
\end{aligned}$$

hold and

$$c(d, e), m, n, r \vdash d$$

does not hold for any mappings m,n or r. The tree t is also matched by the pattern

$$a(\$x, \$y) \in \Phi$$

since

$$a(b, c(d, e)), [x \mapsto b, y \mapsto c(d, e)], \epsilon, \epsilon \vdash a(\$x, \$y)$$

holds and the pattern

$$a(\$xs) \in \Phi$$

since

$$a(b, c(d, e)), \epsilon, [xs \mapsto [b, c(d, e)]], \epsilon \vdash a(\$xs)$$

holds. Note that in the later case the variable $\$xs$ is a placeholder for a forest while in the former case the variables $\$x$ and $\$y$ are placeholders for individual trees. Furthermore, there is no mapping m such that

$$a(b, c(d, e)), m, \epsilon, \epsilon \vdash a(\$x, \$x)$$

would hold since the (sequence) rule demands equivalent variable bindings among its inductive steps. Hence, the pattern $a(\$x, \$x) \in \Phi$ does not match the given tree.

More advanced patterns can be constructed utilizing the rules regarding the binding of variables.

Example 5.5 (advanced tree and list patterns)**.** A pattern $\_(\$a^*)$ where $\$a$ is a tree pattern variable is matching any term by mapping the variable $\$a$ to every individual sub-term of the input term. For instance, it matches the term

$$a(b, c, d)$$

since (among others)

$$\begin{aligned}
a(b, c, d), \epsilon, \epsilon, \epsilon &\vdash \_(\$a^*) && \text{(any node)}\\
[b, c, d], \epsilon, \epsilon, \epsilon &\vdash \$a^* && \text{(repetition)}\\
[c, d], \epsilon, \epsilon, \epsilon &\vdash \$a^* && \text{(repetition)}\\
[b], [a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
[c], [a \mapsto c], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
[d], [a \mapsto d], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
b, [a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(var)}\\
c, [a \mapsto c], \epsilon, \epsilon &\vdash \$a && \text{(var)}\\
d, [a \mapsto d], \epsilon, \epsilon &\vdash \$a && \text{(var)}
\end{aligned}$$

and

$$\begin{aligned}
\epsilon &\sqsubseteq [a \mapsto b]\\
\epsilon &\sqsubseteq [a \mapsto c]\\
\epsilon &\sqsubseteq [a \mapsto d]
\end{aligned}$$

hold. Hence, $a(b, c, d)$ is matched by the pattern $\_(\$a^*)$. However, it is not matched by the almost identical pattern $\_(\$a, \$a^*)$ since the variable $\$a$ is bound to the term b and is then required to be consistent among the succeeding repetitions. When trying to prove a match utilizing

$$\begin{aligned}
a(b, c, d), m, \epsilon, \epsilon &\vdash \_(\$a, \$a^*) && \text{(any node)}\\
[b, c, d], m, \epsilon, \epsilon &\vdash \$a, \$a^* && \text{(repetition)}\\
[b], m[a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
[c, d], m[a \mapsto b], \epsilon, \epsilon &\vdash \$a^* && \text{(repetition)}\\
[c], m[a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
[d], m[a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(single)}\\
c, m[a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(var)}\\
d, m[a \mapsto b], \epsilon, \epsilon &\vdash \$a && \text{(var)}
\end{aligned}$$

the last two, and hence all statements, do not hold for any variable mapping m. Hence, although $\_(\$a, \$a^*)$ seems to be an unrolled version of $\_(\$a^*)$, it does not match the same set of trees. Also since the latter accepts nodes with an empty child list and the former does not.

Finally, the semantic of the recursive pattern constructor shall be illustrated by an example.

Example 5.6 (recursive tree patterns)**.** Let $t = a(a(b)) \in \mathbb{T}$ be a tree. It is matched by the pattern

$$rT.x(b \vee a(rec.x)) \in \Phi$$

since the statements

$$\begin{aligned}
a(a(b)), \epsilon, \epsilon, \epsilon &\vdash rT.x(b \vee a(rec.x)) && \text{(recursion)}\\
a(a(b)), \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash b \vee a(rec.x) && \text{(or)}\\
a(a(b)), \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash a(rec.x) && \text{(node)}\\
a(b), \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash rec.x && \text{(rec. end)}\\
a(b), \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash b \vee a(rec.x) && \text{(or)}\\
a(b), \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash a(rec.x) && \text{(node)}\\
b, \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash rec.x && \text{(rec. end)}\\
b, \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash b \vee a(rec.x) && \text{(or)}\\
b, \epsilon, \epsilon, [x \mapsto (b \vee a(rec.x), \epsilon, \epsilon)] &\vdash b && \text{(constant)}
\end{aligned}$$

hold. A similar prove can not be constructed for e.g. the trees $a(b(a))$ or $a(a(c))$. The tree $a(a(b))$ is also matched by the pattern

$$rT.x(\$y : b \vee a(rec.x)) \in \Phi$$

where each recursively nested sub-tree is bound to the pattern variable $\$y$. Let $c = (\$y : b \vee a(rec.x), \epsilon, \epsilon)$. The validity of

$$a(a(b)), \epsilon, \epsilon, \epsilon \vdash rT.x(\$y : (b \vee a(rec.x)))$$

is demonstrated by the list of valid statements

$$\begin{aligned}
a(a(b)), \epsilon, \epsilon, \epsilon &\vdash rT.x(\$y : (b \vee a(rec.x))) && \text{(recursion)}\\
a(a(b)), [y \mapsto a(a(b))], \epsilon, [x \mapsto c] &\vdash \$y : (b \vee a(rec.x)) && \text{(var)}\\
a(a(b)), [y \mapsto a(a(b))], \epsilon, [x \mapsto c] &\vdash b \vee a(rec.x) && \text{(or)}\\
a(a(b)), [y \mapsto a(a(b))], \epsilon, [x \mapsto c] &\vdash a(rec.x) && \text{(node)}\\
a(b), [y \mapsto a(a(b))], \epsilon, [x \mapsto c] &\vdash rec.x && \text{(rec. end)}\\
a(b), [y \mapsto a(b)], \epsilon, [x \mapsto c] &\vdash \$y : (b \vee a(rec.x)) && \text{(var)}\\
a(b), [y \mapsto a(b)], \epsilon, [x \mapsto c] &\vdash b \vee a(rec.x) && \text{(or)}\\
a(b), [y \mapsto a(b)], \epsilon, [x \mapsto c] &\vdash a(rec.x) && \text{(node)}\\
b, [y \mapsto a(b)], \epsilon, [x \mapsto c] &\vdash rec.x && \text{(rec. end)}\\
b, [y \mapsto b], \epsilon, [x \mapsto c] &\vdash \$y : (b \vee a(rec.x)) && \text{(var)}\\
b, [y \mapsto b], \epsilon, [x \mapsto c] &\vdash b \vee a(rec.x) && \text{(or)}\\
b, [y \mapsto b], \epsilon, [x \mapsto c] &\vdash b && \text{(constant)}
\end{aligned}$$

Variables which have not been bound before entering the processing of the recursive pattern constructor rT are reset upon every recursive instantiation and may therefore be bound to multiple different values. However, within every recursive iteration, the variable binding has to be consistent.

Matches

Besides determining whether a given pattern matches a tree, the list of required variable bindings is equally important for the utilization of patterns within rewriting rules. The structure recording the necessary variable instantiations to make a pattern fit a given tree is referred to as a *variable match.*

Definition 5.9 (variable match)**.** Let $\mathbb{U}_0 = \mathbb{T} \uplus \{\bot\}$ where $\bot$ is the value assigned to unbound variables. Let $\mathbb{U}_{n+1} = \mathbb{U}_n^*$ for all $n \in \mathbb{N}$. The set $\mathbb{U}$ of potential values assigned to variables is given by

$$\mathbb{U} = \bigcup_{0 \leq i} \mathbb{U}_i$$

A *variable match* is a partial mapping $\mathbb{V} \rightharpoonup \mathbb{U}$ assigning values to variables. The set of all variable matches is denoted by $\mathbb{M}$. Furthermore, let $d_\phi : \mathbb{V} \to \mathbb{N}_0$ be the function assigning every variable $x \in \mathbb{V}$ the *repetition depth* of its leftmost, outermost occurrence within a pattern ϕ. Hence, whenever encountering a repetition (ψ^*) or a recursion ($rT.x(\phi)$) along the path from the root of the parse tree of pattern ϕ to the first occurrence of x, the depth is increased by one. Then a variable match $m \in \mathbb{M}$ is valid for a pattern ϕ iff it assigns each variable $x \in \mathbb{V}$ within ϕ an element of $\mathbb{U}_{d_\phi(x)} \cup \{\bot\}$ if x is a tree variable and an element of $\mathbb{U}_{d_\phi(x)+1} \cup \{\bot\}$ if x is a list variable.

Variable matches are obtained from a pattern being matched against a given tree by the *match function.* The match function is a function

$$\text{match} : (\mathbb{T} \times \Phi) \to (\mathbb{M} \cup \{\bot\})$$

such that $\text{match}(t, \phi) = \bot$ in case the given tree $t \in \mathbb{T}$ does not match the pattern $\phi \in \Phi$ and $\text{match}(t, \phi) = m \in \mathbb{M}$ otherwise. In the latter case m is a valid variable match of the pattern ϕ summarizing the necessary variable bindings to prove $t \in T_\phi$.

Thus, the match function needs to verify whether a given tree t matches a pattern ϕ and compute a variable match $m \in \mathbb{M}$ recording the necessary instantiations of the involved variables. For the examples above, the following results need to be obtained:

$$\begin{aligned}
\text{match}(\ a(b,c(d,e))\ ,\ \_\ ) &= \epsilon \\
\text{match}(\ a(b,c(d,e))\ ,\ a(b \vee c, \neg d)\ ) &= \epsilon \\
\text{match}(\ a(b,c(d,e))\ ,\ a(\$x,\$y)\ ) &= [x \mapsto b, y \mapsto c(d,e)] \\
\text{match}(\ a(b,c(d,e))\ ,\ a(\$x,\$x)\ ) &= \perp \\
\text{match}(\ a(b,c(d,e))\ ,\ a(\$xs)\ ) &= [xs \mapsto [b,c(d,e)]] \\
\text{match}(\ a(b,c,d)\ ,\ \_(\$a^*)\ ) &= [a \mapsto [b,c,d]] \\
\text{match}(\ a(a(b))\ ,\ rT.x(b \vee a(rec.x))\ ) &= \epsilon \\
\text{match}(\ a(a(b))\ ,\ rT.x(\$y : (b \vee a(rec.x)))\ ) &= [y \mapsto [a(a(b)),a(b),b]]
\end{aligned}$$

Furthermore, higher dimensional results are obtained for instance by

$$\text{match}(\ a(b(c),b(),b(d,e))\ ,\ a((b(\$x^*))^*)\ ) = [x \mapsto [[c],[],[d,e]]]$$

where the value $[[c],[],[d,e]] \in \mathbb{U}_2$ summarizes the order and the number of times the variable x with repetition depth 2 has to be instantiated to prove that the given tree matches the given pattern.

In the following steps a definition for the match function is developed. A first class of required ingredients are variable match paths.

Definition 5.10 (variable match path)**.** A *variable match path* is a value of the set $\mathcal{P} = \mathbb{N}^*$, hence, a sequence of natural numbers. Further, let the functions $\text{inc} : \mathcal{P} \to \mathcal{P}$ defined by

$$\text{inc}(p) = \begin{cases} [] & \text{if } p = [] \\ [n_1,\ldots,n_k+1] & \text{if } p = [n_1,\ldots,n_k] \end{cases}$$

and the function $\text{push} : \mathcal{P} \to \mathcal{P}$ defined by

$$\text{push}(p) = \text{push}([n_1,\ldots,n_k]) = [n_1,\ldots,n_k,1]$$

be available for manipulating data paths.

Match paths are utilized to address individual values within variable matches. The corresponding operations are to be defined next.

Definition 5.11 (variable match operations)**.** Let $m \in \mathbb{M}$ be a variable match, $p \in \mathcal{P}$ be a variable match path and $x \in \mathbb{V}$ be a pattern variable. Then the value $m_p[x] \in \mathbb{U}$ shall be defined by

$$m_p[x] = \text{get}(m[x], p)$$

where the function $\text{get} : (\mathbb{U} \times \mathcal{P}) \to \mathbb{U}$ is given by

$$\text{get}(u,p) = \begin{cases} u & \text{if } p = [] \\ \text{get}(u_{n_1},[n_2,\ldots,n_k]) & \text{if } p = [n_1,\ldots,n_k] \text{ and } |u| \leq n_1 \\ \perp & \text{otherwise} \end{cases}$$

Furthermore, let $v \in \mathbb{U}$ be a value. Then, the $m_p[x] := v$ denotes a modified variable mapping defined by

$$(m_p[x] := v) = m[x \mapsto \text{set}(m[x], p, v)]$$

where the function set : $(\mathbb{U} \times \mathcal{P} \times \mathbb{U}) \to \mathbb{U}$ is given by

$$\text{set}(u,p,v) = \begin{cases} v & \text{if } p = [] \\ [\text{set}'(u,1,n_1,v'), \dots, \text{set}'(u, \max(k,l), n_1, v')] & \\ & \text{if } u = [u_1, \dots, u_k] \text{ and } p = [n_1, \dots, n_l] \\ & \quad \text{and } v' = \text{set}(u_{n_1}, [n_2, \dots, n_l], v) \end{cases}$$

and the function set$'$: $(\mathbb{U}^* \times \mathbb{N} \times \mathbb{N} \times \mathbb{U}) \to \mathbb{U}$ by

$$\text{set}'(u,i,n,v) = \begin{cases} u_i & \text{if } i \neq n \text{ and } |u| \leq i \\ v & \text{if } i = n \\ \perp & \text{otherwise} \end{cases}$$

Hence, $m_p[x] := v$ is the mapping where the element addressed by p within the value assigned by m to x is replaced by the value v.

To illustrate the utilization of match paths and match operations a few examples shall be provided.

Example 5.7 (match operations)**.** Let

$$m = [x \mapsto a, y \mapsto [b,c]] \in \mathbb{M}$$

be a variable mapping. Then $m_{[]}[x] = a$, $m_{[]}[y] = [b,c]$, $m_{[1]}[y] = b$, $m_{[2]}[y] = c$ and $m_{[3]}[y] = \perp$. Hence, the data path in the subscript can be utilized to address and obtain an element of the nested value assigned to a given variable. Furthermore, the term $m_{[2]}[y] := d$ corresponds to the mapping

$$[x \mapsto a, y \mapsto [b,d]] \in \mathbb{M}$$

$m_{[]}[y] := d$ to

$$[x \mapsto a, y \mapsto d] \in \mathbb{M}$$

$m_{[5]}[y] := d$ to

$$[x \mapsto a, y \mapsto [b,c,\perp,\perp,d]] \in \mathbb{M}$$

and $m_{[]}[z] := d$ to

$$[x \mapsto a, y \mapsto [b,c], z \mapsto d] \in \mathbb{M}$$

Thus, this construct can be utilized to modify and extend variable matches.

Based on match paths and operators the match function can be defined as follows.

Definition 5.12 (match function)**.** Let $\mathbb{M}_\bot = \mathbb{M} \uplus \{\bot\}$. Further, let

$$\mathcal{D} = (\Phi \times \mathbb{T} \times \mathcal{P}) \uplus (\Psi \times \mathbb{T}^* \times \mathcal{P})$$

be a set of pattern/structure/path triples and

$$\mathcal{R} = \mathbb{V} \rightharpoonup (\Phi \times \mathcal{P})$$

the set of partial mappings mapping variables to pairs of tree patterns and match paths. Then the function

$$\text{match} : (\mathbb{T} \times \Phi) \to \mathbb{M}_\bot$$

is defined by

$$\text{match}(t, \phi) = \text{match}'([(\phi, t, [])], \epsilon)$$

where the function $\text{match}' : (\mathcal{D}^* \times \mathbb{M}_\bot) \to \mathbb{M}_\bot$ is defined by

$$\text{match}'(\vec{d}, m) = \begin{cases} \bot & \text{if } m = \bot \\ m & \text{if } \vec{d} = [] \\ m_\Phi(\phi, t, p, \vec{r}, m) & \text{if } \vec{d} = [(\phi, t, p), \vec{r}] \wedge \phi \in \Phi \\ m_\Psi(\psi, \vec{s}, p, \vec{r}, m) & \text{if } \vec{d} = [(\psi, \vec{s}, p), \vec{r}] \wedge \psi \in \Psi \end{cases}$$

where the function $m_\Phi : (\Phi \times \mathbb{T} \times \mathcal{P} \times \mathcal{D}^* \times \mathbb{M}_\bot \times \mathcal{R}) \to \mathbb{M}_\bot$ is given by

$$m_\Phi(\phi, t, p, \vec{d}, m, r) =$$

$$\begin{cases}
\bot & \text{if } m = \bot \\
\mathrm{match}'(\vec{d}, m) & \text{if } \phi = \_ \\
\mathrm{match}'(\vec{d}, m) & \text{if } \phi = t \in \mathbb{T} \\
m & \text{if } \phi = \neg\phi_1 \text{ and } m_\Phi(\phi_1, t, p, \vec{d}, m, r) = \bot \\
\bot & \text{if } \phi = \neg\phi_1 \text{ and } m_\Phi(\phi_1, t, p, \vec{d}, m, r) \neq \bot \\
m_\Phi(\phi_1, t, p, [(\phi_2, t, p), \vec{d}], m, r) & \\
 & \text{if } \phi = \phi_1 \wedge \phi_2 \\
m_\Phi(\phi_1, t, p, \vec{d}, m, r) & \text{if } \phi = \phi_1 \vee \phi_2 \text{ and } m_\Phi(\phi_1, t, p, \vec{d}, m, r) \neq \bot \\
m_\Phi(\phi_2, t, p, \vec{d}, m, r) & \text{if } \phi = \phi_1 \vee \phi_2 \text{ and } m_\Phi(\phi_1, t, p, \vec{d}, m, r) = \bot \\
m_\Psi(\psi, \vec{s}, p, \vec{d}, m, r) & \text{if } \phi = \_(\psi) \text{ and } t = k(\vec{s}) \\
m_\Psi(\psi, \vec{s}, p, \vec{d}, m, r) & \text{if } \phi = k(\psi) \text{ and } t = k(\vec{s}) \\
\bot & \text{if } \phi = k(\psi) \text{ and } t = k'(\vec{s}) \text{ and } k \neq k' \\
m_\Phi(\phi_1, t, p, \vec{d}, m_p[x] := t, r) & \\
 & \text{if } \phi = x : \phi_1 \text{ and } m_p[x] = \bot \\
\mathrm{match}'(\vec{d}, m) & \text{if } \phi = x : \phi_1 \text{ and } m_p[x] = t \\
\bot & \text{if } \phi = x : \phi_1 \text{ and } m_p[x] \notin \{\bot, t\} \\
m_\Phi(\phi_1 \vee \_(\_^*, \phi, \_^*), t, p, \vec{d}, m, r) & \\
 & \text{if } \phi = aT(\phi_1) \\
m_\Phi(\phi_1, t, \mathrm{push}(p), \vec{d}, m, r[x \mapsto (\phi_1, \mathrm{push}(p))]) & \\
 & \text{if } \phi = rT.x(\phi_1) \\
m_\Phi(\phi_1, t, \mathrm{inc}(p_1), \vec{d}, m, r[x \mapsto (\phi_1, \mathrm{inc}(p_1))]) & \\
 & \text{if } \phi = rec.x \text{ and } r[x] = (\phi_1, p_1) \\
\bot & \text{otherwise}
\end{cases}$$

and the function $m_\Psi : (\Psi \times \mathbb{T}^* \times \mathcal{P} \times \mathcal{D}^* \times \mathbb{M}_\bot \times \mathcal{R}) \to \mathbb{M}_\bot$ is given by

$m_\Psi(\psi, \vec{s}, p, \vec{d}, m, r) =$

$$\begin{cases}
\bot & \text{if } m = \bot \\
\mathrm{match}'(\vec{d}, m) & \text{if } \psi = \epsilon \text{ and } \vec{s} = [] \\
m_\Phi(\phi, t, p, \vec{d}, m, r) & \text{if } \psi = \phi \in \Phi \text{ and } \vec{s} = [t] \\
m' & \text{if } \psi = \psi_1, \psi_2 \text{ and } \exists \vec{s_1}, \vec{s_2} \in \mathbb{T}^* \;.\; (\vec{s} = [\vec{s_1}, \vec{s_2}] \wedge \\
 & \quad m_\Psi(\psi_1, \vec{s_1}, p, [(\psi_2, \vec{s_2}, p), \vec{d}], m, r) = m' \neq \bot) \\
\bot & \text{if } \psi = \psi_1, \psi_2 \text{ and } \nexists \vec{s_1}, \vec{s_2} \in \mathbb{T}^* \;.\; (\vec{s} = [\vec{s_1}, \vec{s_2}] \wedge \\
 & \quad m_\Psi(\psi_1, \vec{s_1}, p, [(\psi_2, \vec{s_2}, p), \vec{d}], m, r) \neq \bot) \\
m_\Psi(\psi_1, \vec{s}, p, \vec{d}, m, r) & \text{if } \psi = \psi_1 \vee \psi_2 \text{ and } m_\Psi(\psi_1, \vec{s}, p, \vec{d}, m, r) \neq \bot \\
m_\Psi(\psi_2, \vec{s}, p, \vec{d}, m, r) & \text{if } \psi = \psi_1 \vee \psi_2 \text{ and } m_\Psi(\psi_1, \vec{s}, p, \vec{d}, m, r) = \bot \\
m_\Psi(\psi_1, \vec{s}, p, \vec{d}, m_p[x] := \vec{s}, r) & \\
 & \text{if } \psi = x : \psi_1 \text{ and } m_p[x] = \bot \\
\mathrm{match}'(\vec{d}, m) & \text{if } \phi = x : \psi_1 \text{ and } m_p[x] = \vec{s} \\
\bot & \text{if } \phi = x : \psi_1 \text{ and } m_p[x] \notin \{\bot, \vec{s}\} \\
m_*(\psi_1, \vec{s}, \mathrm{push}(p), \vec{d}, m, r) & \\
 & \text{if } \psi = \psi_1^* \\
\bot & \text{otherwise}
\end{cases}$$

and the function $m_* : (\Psi \times \mathbb{T}^* \times \mathcal{P} \times \mathcal{D}^* \times \mathbb{M}_\perp \times \mathcal{R}) \to \mathbb{M}_\perp$ is defined by

$$m_*(\psi, \vec{s}, p, \vec{d}, m, r) =$$
$$\begin{cases} \perp & \text{if } m = \perp \\ \text{match}'(\vec{d}, m) & \text{if } \vec{s} = [] \\ m' & \text{if } \exists l_1, l_2 \in \mathbb{T}^* \, . \, (\vec{s} = [\vec{s_1}, \vec{s_2}] \wedge \\ & \quad m_*(\psi, \vec{s_2}, \text{inc}(p), \vec{d}, m_\Psi(\psi, \vec{s_1}, p, [], m, r), r) = m' \neq \perp) \\ \perp & \text{otherwise} \end{cases}$$

where inc and push are the operators defined on match paths.

The basic idea of the given match function definition is to gradually match sub-patterns against sub-structures of the matched tree. This list of pending match operations is forwarded through the computation utilizing the parameter $\vec{d} \in \mathcal{D}^*$, extended if required, and stepwise consumed by the definition of the function match$'$. During the course of processing sub-problems by the functions m_Φ and m_Ψ, variable bindings are accumulated by the match result passed along as the parameter $m \in \mathbb{M}_\perp$. The parameter $p \in \mathcal{P}$ is utilized to keep track of the repetition-depth and the number of repetitions on the various levels. Finally, the parameter $r \in \mathcal{R}$ is utilized to maintain recursive variable bindings.

An implementation of the presented match function and a variety of examples in the form of test cases are included in the Insieme sources (see Appendix A).

Replacements

A replacement expression is a term representing a script turning a matched tree $t_m \in \mathbb{T}$ and a variable matching $m \in \mathbb{M}$ into a new tree to be utilized to substitute t_m.

Definition 5.13 (generator syntax)**.** For the definition of replacement expressions three types of generator expressions are distinguished: expressions producing results of type $\mathbb{T}$ (tree generators, τ), results of type $\mathbb{T}^*$ (list generators, σ), and results of type $\mathbb{U}$ (value generators, υ). Their structure is defined by the three production rules

$$\begin{aligned} \tau &::= \upsilon \mid k\,(\sigma) \mid \tau\,[\tau/\tau] \\ \sigma &::= \upsilon \mid \epsilon \mid [\tau] \mid \sigma, \sigma \\ \upsilon &::= \lambda_c \mid \lambda_t\,(\upsilon) \mid \tau \mid \sigma \mid \textit{let } x = \upsilon \textit{ in } \upsilon \mid \forall x \in \upsilon \, . \, \upsilon \end{aligned}$$

where $k \in \mathbb{K}$ is a node type, $\lambda_c : (\mathbb{T} \times \mathbb{M}) \to \mathbb{U}$ is a function creating values, $\lambda_t : \mathbb{U} \to \mathbb{U}$ is a function transforming values and $x \in \mathbb{V}$ is a variable. Let Δ, Σ and Υ denote the set of expressions being generated by τ, σ and υ respectively.

Definition 5.14 (generator semantic)**.** The semantics of our generator expressions are given by the three functions

$$\begin{aligned}\Gamma_\Delta &: \mathbb{T} \times \mathbb{M} \times \Delta \to \mathbb{T}\\ \Gamma_\Sigma &: \mathbb{T} \times \mathbb{M} \times \Sigma \to \mathbb{T}^*\\ \Gamma_\Upsilon &: \mathbb{T} \times \mathbb{M} \times \Upsilon \to \mathbb{U}\end{aligned}$$

defined by

$$\Gamma_\Delta(t,m,\tau) = \begin{cases} \Gamma_\Upsilon(t,m,\upsilon) & \text{if } \tau \text{ is } \upsilon \text{ and } \Gamma_\Upsilon(t,m,\upsilon) \in \mathbb{T} & (expr)\\ k\left(\Gamma_\Sigma(t,m,\sigma)\right) & \text{if } \tau \text{ is } k(\sigma) & (node)\\ let\ t_1 = \Gamma_\Delta(t,m,\tau_1)\ in & & \\ let\ t_2 = \Gamma_\Delta(t,m,\tau_2)\ in & & \\ let\ t_3 = \Gamma_\Delta(t,m,\tau_3)\ in & & \\ t_1[t_2/t_3] & \text{if } \tau \text{ is } \tau_1[\tau_2/\tau_3] & (substitution) \end{cases}$$

$$\Gamma_\Sigma(t,m,\sigma) = \begin{cases} \Gamma_\Upsilon(t,m,\upsilon) & \text{if } \sigma \text{ is } \upsilon \text{ and } \Gamma_\Upsilon(t,m,\upsilon) \in \mathbb{T}^* & (expr)\\ \epsilon & \text{if } \sigma \text{ is } \epsilon & (empty)\\ \left[\Gamma_\Delta(t,m,\tau)\right] & \text{if } \sigma \text{ is } [\tau] & (single)\\ \Gamma_\Sigma(t,m,\sigma_1), \Gamma_\Sigma(t,m,\sigma_2) & \text{if } \sigma \text{ is } \sigma_1,\sigma_2 & (sequence) \end{cases}$$

$$\Gamma_\Upsilon(t,m,\upsilon) = \begin{cases} \lambda_c(t,m) & \text{if } \upsilon \text{ is } \lambda_c & (ctor)\\ \lambda_t\left(\Gamma_\Upsilon(t,m,\upsilon')\right) & \text{if } \upsilon \text{ is } \lambda_t(\upsilon') & (transform)\\ \Gamma_\Delta(t,m,\tau) & \text{if } \upsilon \text{ is } \tau & (tree)\\ \Gamma_\Sigma(t,m,\sigma) & \text{if } \upsilon \text{ is } \sigma & (list)\\ \Gamma_\Upsilon\left(t, m[x \mapsto \Gamma_\Upsilon(t,m,\upsilon_1)], \upsilon_2\right) & \text{if } \upsilon \text{ is } let\ x = \upsilon_1\ in\ \upsilon_2 & (let)\\ let\ [u_1,\ldots,u_n] = \Gamma_\Upsilon(t,m,\upsilon_1)\ in & & \\ let\ f = \lambda y.\Gamma_\Upsilon(t, m[x \mapsto y], \upsilon_2) & & \\ in\ [f(u_1),\ldots,f(u_n)] & \text{if } \upsilon \text{ is } \forall x \in \upsilon_1\ .\ \upsilon_2 & (foreach) \end{cases}$$

where $t_1[t_2/t_3] \in \mathbb{T}$ denotes the tree obtained by replacing all occurrences of $t_3 \in \mathbb{T}$ within $t_1 \in \mathbb{T}$ by $t_2 \in \mathbb{T}$.

Example 5.8 (generators)**.** Let $m = [x \mapsto [a,b,c]] \in \mathbb{M}$ be the result of matching a pattern on a given tree t. We would like to produce a tree with a root node of kind r and the elements mapped to x forming the child list. Such a tree generator expression can be represented by

$$r(\lambda_x) \in \Delta$$

utilizing the production rules $\tau ::= k(\sigma)$, $\sigma ::= \upsilon$ and $\upsilon ::= \lambda_c$ where $k = r$, $\lambda_c = \lambda_x$ and λ_x is defined by

$$\lambda_x(t,m) = m[x]$$

Based on those we obtain

$$\begin{aligned}\Gamma_\Delta(t,m,r(\lambda_x)) &= r(\Gamma_\Sigma(t,m,\lambda_x))\\ &= r(\Gamma_\Upsilon(t,m,\lambda_x))\\ &= r(\lambda_x(t,m))\\ &= r(m[x])\\ &= r(a,b,c) \in \mathbb{T}\end{aligned}$$

as desired. If, for instance, $r(s(a,a), s(b,b), s(c,c))$ shall be produced instead, the tree generator expression

$$r(\forall y \in \lambda_x \,.\, s([\lambda_y], [\lambda_y])) \in \Delta$$

can be utilized where $\lambda_y(t,m) = m[y]$. Computing

$$\begin{aligned}
&\Gamma_\Delta(t, m, r(\forall y \in \lambda_x \,.\, s([\lambda_y], [\lambda_y])))\\
&\quad = r(\Gamma_\Sigma(t, m, \forall y \in \lambda_x \,.\, s([\lambda_y], [\lambda_y])))\\
&\quad = r([\\
&\qquad\quad \Gamma_\Upsilon(t, m[y \mapsto a], s([\lambda_y], [\lambda_y])),\\
&\qquad\quad \Gamma_\Upsilon(t, m[y \mapsto b], s([\lambda_y], [\lambda_y])),\\
&\qquad\quad \Gamma_\Upsilon(t, m[y \mapsto c], s([\lambda_y], [\lambda_y]))\\
&\qquad])\\
&\quad = r([\\
&\qquad\quad s(\Gamma_\Sigma(t, m[y \mapsto a], [\lambda_y]), \Gamma_\Sigma(t, m[y \mapsto a], [\lambda_y])),\\
&\qquad\quad s(\Gamma_\Sigma(t, m[y \mapsto b], [\lambda_y]), \Gamma_\Sigma(t, m[y \mapsto b], [\lambda_y])),\\
&\qquad\quad s(\Gamma_\Sigma(t, m[y \mapsto c], [\lambda_y]), \Gamma_\Sigma(t, m[y \mapsto c], [\lambda_y]))\\
&\qquad])\\
&\quad = r([\\
&\qquad\quad s([\Gamma_\Delta(t, m[y \mapsto a], \lambda_y)], [\Gamma_\Delta(t, m[y \mapsto a], \lambda_y)]),\\
&\qquad\quad s([\Gamma_\Delta(t, m[y \mapsto b], \lambda_y)], [\Gamma_\Delta(t, m[y \mapsto b], \lambda_y)]),\\
&\qquad\quad s([\Gamma_\Delta(t, m[y \mapsto c], \lambda_y)], [\Gamma_\Delta(t, m[y \mapsto c], \lambda_y)])\\
&\qquad])\\
&\quad = r(s(a,a), s(b,b), s(c,c)) \in \mathbb{T}
\end{aligned}$$

yields the desired result. Note that in this calculation several steps, in particular those switching between the functions Γ_Δ, Γ_Σ and Γ_Υ, have been omitted for brevity.

Bringing it all together: Rules

Finally, pattern and generator expression can be combined into *rules* representing high-level descriptions of code transformations.

Definition 5.15 (rule)**.** A *rule* is a pair $(\phi, \tau) \in \Phi \times \Delta$, denoted by $\phi \to \tau$, where $\phi \in \Phi$ is a tree pattern and $\tau \in \Delta$ is a tree generator expression.

When applying a rule on a tree $t \in \mathbb{T}$ it is determined whether there are mappings m and n such that $t, m, n, \emptyset \vdash \phi$. If so, a variable match $\text{match}(t, \phi) = m' \in \mathbb{M}$ is computed and utilized to generate the replacement $\Gamma_\Delta(t, m', \tau) \in \mathbb{T}$.

Example 5.9 (rules)**.** Let us consider a rule removing all immediate sub-terms of the shape $a(\_^*)$ from a given term of kind r. For instance,

$$r(a(), b, c(d, e), a(c), d)$$

should become

$$r(b, c(d, e), d)$$

A pair of adequate pattern and generator expressions has to be devised such that all the necessary information to construct the result is captured. An example solution would be

$$\begin{aligned}\phi &= r(\$xs : (\neg a(\_^*))^*, (a(\_^*), \$ys : (\neg a(\_^*))^*)^*) \\ \tau &= r(\$xs, (\forall y \in \lambda_{ys} \,.\, \$y))\end{aligned}$$

where $\lambda_{ys}(t, m) = m[ys]$. The pattern ϕ identifies sequences of sub-terms in the root-term $r(\ldots)$ not containing any element of the shape $a(\_^*)$ utilizing the pattern $(\neg a(\_^*))^*$. The sequence before the first occurrence of an a-term is recorded by the list variable xs, sequences between a-terms and between the last a-term and the end by the variable ys. Note that while xs is a list variable with repetition-depth $d_\phi(xs) = 0$ the variable ys has repetition-depth $d_\phi(ys) = 1$. Hence, within matches, xs will be associated to list of trees while ys is associated to lists of lists of trees. The generator pattern τ is simply composing the obtained fragments to the desired result by concatenating the elements marked by xs with the concatenation of the elements associated to ys. When applying the rule $\phi \to \tau$ on the term

$$t = r(a(), b, c(d, e), a(c), d)$$

we obtain

$$m = \text{match}(t, \phi) = [xs \mapsto [], ys \mapsto [[b, c(d, e)], [d]]] \in \mathbb{M}$$

and

$$\Gamma_\Delta(t, m, \tau) = r(b, c(d, e), d) \in \mathbb{T}$$

as desired. Applied to $t_2 = r(a, b, c)$ it yields

$$m_2 = \text{match}(t_2, \phi) = [xs \mapsto [a, b, c], ys \mapsto []] \in \mathbb{M}$$

and $\Gamma_\Delta(t_2, m_2, \tau) = r(a, b, c) \in \mathbb{T}$ and on $t_3 = r(a(a, b, c))$ we obtain

$$m_3 = \text{match}(t_3, \phi) = [xs \mapsto [], ys \mapsto []] \in \mathbb{M}$$

and $\Gamma_\Delta(t_3, m_3, \tau) = r() \in \mathbb{T}$.

For both, the pattern and the generator constructs, the presented constructs merely present the core structures establishing the formal foundation for the pattern matching and replacement generation processes. Those core constructs are wrapped up into domain specific derived constructs before being utilized by the end user. Examples of those are covered in the following section.

5.3.3 Implementation

Like the rest of the Insieme Compiler infrastructure, the implementation of the pattern based transformation toolbox is based on C++11. It represents patterns and replacements as objects, with common connectors mapped to overloaded C++ operators. This approach has multiple advantages: *composability* (simple pattern composition using C++ variables, operators and functions), *extensibility* (new user-defined constructs may be added), *reliability* (pattern fragments can be tested independently using common unit testing frameworks, and their types and arities are checked at compile time), and *productivity* (all integrated development environment features including code completion apply to patterns, and users are not required to learn a new language syntax). The examples in Section 5.3.4 illustrate some of these advantages.

Due to overloading, multiple variations of the same pattern construct can be offered, which is particularly useful when defining derived constructs dealing with IR primitives. For instance, there might be two overloaded functions

```
TreePattern forStmt() {
  return /* some k(..) primitive with wildcards */
}
TreePattern forStmt(const TreePattern& body) {
  return /* some k(..) primitive including body */
}
```

where the first creates a pattern matching any for loop while the latter allows the user to constrain the body of a matched loop. Both utilize the $\phi ::= k(\psi)$ production rule where k is replaced by the token identifying for-statement nodes and ψ by a correspondingly composed child-node list pattern. Based on those, the statements

```
auto noFor = !forStmt();
auto p = forStmt(noFor) | forStmt(forStmt(noFor));
```

create a pattern p matching a single for loop or two perfectly nested loops. Note the utilization of C++ variables, the derived, domain specific constructs $forStmt$ and the overloaded C++ negation operator ! and disjunction operator | for composing the desired pattern in a comprehensible and concise way. Like for the $forStmt$, similar, partially overloaded constructors are offered for all kind of IR nodes.

The Matching Algorithm

The centerpiece of our system is the pattern matching algorithm. In our implementation it is based on a back-tracking approach following Definition 5.12. While for most primitives the check whether a given tree matches the corresponding pattern is straightforward (e.g. the constant t, wildcards, negations and conjunctions) the processing of a few primitives requires more sophisticated steps. For instance, to determine whether a forest s satisfies a pattern ψ_1, ψ_2 the sequence s needs to be split up into two sub-sequences s_1 and s_2 (see Definition 5.12). However, the splitting point can only be guessed at this point – and in case it was wrong altered in a back-tracking step.

To increase the probability of guessing right as soon as possible we employ several pruning heuristics. For instance, if a sequence pattern of the shape $\psi_1, \psi_2, \ldots, \psi_n$ contains constant tree patterns ($\psi_i = t$) or patterns demanding a fixed node type (e.g. $\psi_i = k(\ldots)$) these elements are identified within a potential candidate list $t_1, \ldots, t_m$ before the remaining, potentially more complex patterns are matched against the interjacent sub-sequences. Also, memoization is utilized to avoid resolving identical sub-problems multiple times.

Nevertheless, the worst case complexity of our matching algorithm is exponential. However, so far, for real-world patterns encountered within our daily interaction with the system the search space pruning heuristics are effective enough to not impose a substantial performance issue. Also, the system benefits from the limited average length of sequences encountered within ASTs.

5.3.4 Examples

In general, when defining sophisticated patterns the definition of auxiliary connectors is beneficial. Therefore, in addition to the constants any (_), anyList (_*), the primitive connector node(...) matching any node with a given child list and the derived connector step(a), which is equivalent to node(*any << a << *any) where the << operator is the sequence connector of list patterns, we have defined the following constructs

```
TreePattern all(const TreePattern& a) {
  return rT((a & node(*rec)) | (!a & node(*rec)));
}
TreePattern outermost(const TreePattern& a) {
  return rT(a | (!a & node(*rec)));
}
TreePattern innermost(const TreePattern& a) {
  return rT((!step(aT(a)) & a) | node(*rec));
}
```

The derived constructs all, outermost and innermost collect all / the outermost / the innermost sub-trees matching a given input pattern. Based on those, patterns locating loops at corresponding positions can be constructed by

```
auto a = all(var("x", forStmt()));
auto b = outermost(var("x", forStmt()));
auto c = innermost(var("x", forStmt()));
```

Also, a pattern identifying a used variable and all its accesses is created by

```
auto x = var("x");
auto use = !declStmt() & step(x);
auto p = aT(declStmt(x)) & aT(use)
        & all(var("y",use));
```

The resulting pattern p checks for the presence of a declaration defining a utilized variable x and collects all its uses within the pattern variable y.

Aggregating Operators

As an example seen from the developer's perspective, let us consider the design of a transformation converting the consecutive application of a multiplication and an addition into a single application of a combined *multiply-and-add* operator. Hence, the expressions

```
a * b + c;
4 * (2-c) + d;
12 * (2*a+1) + (a*b+3);
```

shall be transformed into

```
mad(a,b,c);
mad(4,2-c,d);
mad(12,mad(2,a,1),mad(a,b,3));
```

Let *call* be an overloaded C++ function creating patterns for call expression nodes and the constants *mul*, *add* and *mad* be constants for the multiplication, addition and multiply-and-add operators. The corresponding transformation rule is created by

```
Variable t = "t";
Variable a = "a";
Variable b = "b";
Variable c = "c";

auto p = call(t, add, call(mul, a << b) << c);
auto g = call(t, mad, a << b << c);
auto r = Rule(p, g);
```

where the variable t is utilized to capture the type of the expression and the variables a, b and c the sub-trees constituting the operands. The resulting rule r can be applied directly on an expression and, if matching, would conduct the desired transformation. However, it does not handle cases in which the targeted case is nested within another term. To add support for nested cases the given pattern can be extended by

```
Variable trg("i",p);
auto p2 = aT(trg);
auto g2 = substitute(root, trg, g);
auto r2 = Rule(p2,g2);
```

such that $p2$ is searching for a term matching p within a given tree and $g2$ replaces the located sub-term marked by the variable trg by the replacement produced by the generator expression g. Thus, the resulting rule $r2$ is capable of locating a nested term fitting the pattern $a * b + c$ and replacing it by an application of the multiply-and-add operator. Each application of the rule on a code fragment *in* using

```
auto out = r2(in);
```

conducts a single replacement. Repeatedly applying the rule $r2$ until no more modifications are conducted, hence until a fixpoint is reached, using

```
auto out = r2.fixpoint(in);
```

results in a code fragment *out* where all fitting sub-expressions have been substituted by an application of the *mad* operator.

By utilizing C++ facilities including overloaded functions and operators as well as a functional programming style, the complex, yet concise nature of the core primitives of the pattern matcher and generator utilities are effectively concealed without losing expressiveness nor flexibility.

A Real-World Transformation

To demonstrate the applicability of the pattern based transformation system to more complex tasks, a transformation designed to eliminate redundant sync calls within Cilk applications, which was used in the context of research conducted based on the Insieme infrastructure [99], shall be presented. A sync is deemed redundant if there has not been any spawn invocation since the last sync.

Let spawn and sync be constants representing patterns matching applications of the corresponding operators. In a first step the pattern

```
auto unsynced =
  rT(spawn | node(*any << aT(rec) << *!sync));
```

matching code fragments potentially resulting in an unsynchronized state is defined. This is the case for a plain spawn statement or whenever an arbitrarily nested spawn is not followed by a sync on the same or an enclosing scope. In the next step the predicate unsynced is utilized to define the predicate synced as well as the desired pattern p identifying redundant sync statements:

```
auto synced = !unsynced;
auto p = compound(
  opt(*any << sync) << *synced << var("x", sync) << *any
);
```

The pattern p searches code fragments in which an optional (opt(..)) safe sequence of explicitly (on the same scope) or implicitly (nested) synchronized statements is followed by a sync call, which gets bound to variable x. The replacement expression

```
auto r = substitute(root, var("x"), noop);
```

can be utilized to generate a proper substitute for any matched statement. Here root is the root of the matched sub-tree, var("x") extracts the value bound to variable x and noop is a constant representing an expression conducting no operation. It replaces the identified redundant sync call x by a noop. Finally, the following code fragment combines p and r into a (transformation) rule and applies it to a target-code fragment in:

```
Rule syncElimination = Rule(p,r);
auto out = syncElimination(in);
```

The application of the rule includes the computation of a match result for pattern p and forwards it to the replacement expression r to produce the desired result.

5.4 Polyhedral Transformations

Besides its analytic capabilities, the polyhedral model also provides a powerful foundation for code transformations – in particular targeting loop nests. Contemporary compiler infrastructures are frequently equipped with support for such transformations. The basics of those as well as their integration into the Insieme compiler infrastructure shall be briefly outlined within this section.

5.4.1 Overview on Polyhedral Transformations

A simple way to utilize the polyhedral model is to conduct dependency checks on loop nests as covered in Section 4.5 to verify whether loop transformations can be applied without altering the observable semantic of the targeted code fragment. The actual modification may then be realized utilizing node mapper or pattern based transformations.

However, the polyhedral model itself offers a much more powerful basis for conducting transformations [12]. Given the code fragment

```
let int = int<4>;
for(int i = 0 .. N) {
  for(int j = i .. N) {
    ... = a[i][j-1];            // S1
    a[i][j] := ...;             // S2
  }
}
```

and the polyhedral description

$$\left(\vec{I}, \left\{ \left(\mathcal{D}_{S1}, \mathcal{T}_{S1}, \left\{\mathcal{A}_{(\text{S1,USE},a)}\right\}\right), \left(\mathcal{D}_{S2}, \mathcal{T}_{S2}, \left\{\mathcal{A}_{(\text{S2,DEF},a)}\right\}\right) \right\}\right)$$

obtained in Section 4.5.1 the order of execution of the involved statements is covered by the *schedule functions* represented by the matrices $\mathcal{T}_{S1}$ and $\mathcal{T}_{S2}$ for each of the involved statements. It also covers the access to the involved arrays and – implicitly – their number of dimensions and the order of their indices, hence their memory layout. Effectively, the polyhedral model offers a code representation enabling the isolated consideration of the scheduling of instruction instances and the associated memory accesses. Altering those in the polyhedral representation turns out to be much more efficient, scalable and composable than equivalent transformations utilizing AST-based operations [12]. The only requirement this imposes is a component capable of converting an arbitrary polyhedral description back int a AST-based structure, hence proper IR code.

Fortunately, such an algorithm exist and is implement e.g. by the *Chunky Loop Generator* (GLooG) library [13]. Thus, arbitrary modifications preserving the dependencies of the original input code can be applied on the polyhedral representation of a code fragment before converting it back into IR code. Modifications on the schedule functions may thereby have similar effects in a single step as long sequences of conventional textbook loop transformations including e.g. loop fission, fusion, permutation, reversal skewing and tiling. Consistently, altering access functions allows re-shaping the memory layout of arrays as well as access patterns. The allowed modifications, however, are limited under the constraints of the involved data dependencies which can be accurately analyzed utilizing the same representation.

In particular, scheduling and optimization problems based on the polyhedral model can be represented as (linear) optimization problems tuning a given objective function, e.g. maximizing data reuse [87]. Effectively, the problem of obtaining optimal code transformations is converted into solving a mathematical optimization problem for which sophisticated optimization techniques can be utilized.

Example Transformation

As has been obtained in Section 4.5.1, the code fragment above contains a read-after-write dependency preventing e.g. the inner loop from being parallelized. By reordering the two statements and splitting the inner loop, this dependency could be resolved. The resulting code would be similar to

```
let int = int<4>;
for(int i = 0 .. N) {
   for(int j = i .. N) {
      a[i][j] := ...;                   // S2
   }
   for(int j = i .. N) {
      ... = a[i][j-1];                  // S1
   }
}
```

Those two transformation steps can be conducted in a single step utilizing the polyhedral model. For the given example, the two matrices $\mathcal{T}_{S1}$ and $\mathcal{T}_{S2}$ given by

$$\mathcal{T}_{S1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{and} \qquad \mathcal{T}_{S2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

determine the order in which instances of statements $S1$ and $S2$ are processed. A generic instance $I = (i, j, N, 1)^T$ of $S1$ is executed at timestamp $\mathcal{T}_{S1}I = (i, j, 0)^T$ and instance I of statement $S2$ at the logical timestamp $\mathcal{T}_{S2}I = (i, j, 1)^T$. However, in the transformed code we would like all instances of $S2$ associated to the inner loop being processed before the corresponding instances of $S1$. This is, for instance, achieved by obtaining modified matrices $\mathcal{T}'_{S1}$ and $\mathcal{T}'_{S2}$ such that

$$\mathcal{T}'_{S1}I = (i, 1, j, 0)^T \qquad \text{and} \qquad \mathcal{T}'_{S2}I = (i, 0, j, 1)^T$$

which corresponds to the matrices

$$\mathcal{T}'_{S1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{and} \qquad \mathcal{T}'_{S2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the resulting lexicographical order corresponds to the desired effect and that, compared to the original matrices, only an additional row had to be introduced in both matrices. The resulting representation can then be used to check whether all dependencies encountered in the original code fragment are still valid – utilizing PM based analyses – to verify the validity of the applied transformation. Finally, by substituting the original scheduling matrices with their modified versions and converting the resulting polyhedral description

$$\left(\vec{I}, \left\{\left(\mathcal{D}_{S1}, \mathcal{T}'_{S1}, \left\{\mathcal{A}_{(\mathrm{S1,USE},a)}\right\}\right), \left(\mathcal{D}_{S2}, \mathcal{T}'_{S2}, \left\{\mathcal{A}_{(\mathrm{S2,DEF},a)}\right\}\right)\right\}\right)$$

back into AST like code, a result equivalent to the code fragment outlined above is obtained.

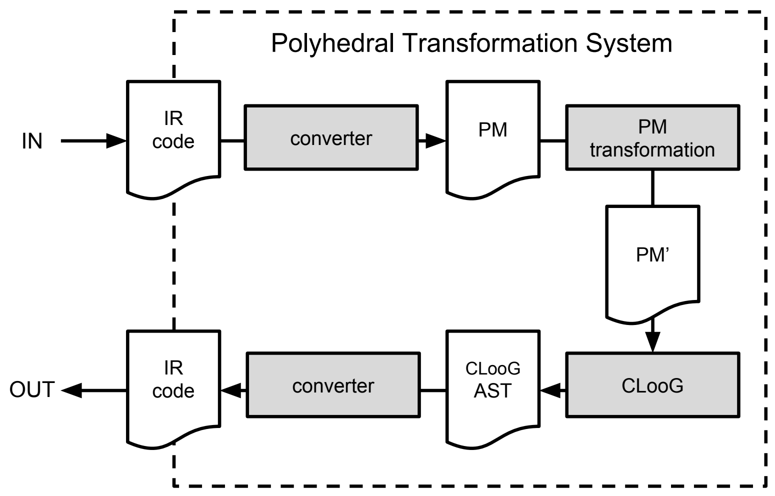


Figure 5.1: Overview on the polyhedral transformation system.

However, this example represents only a simple use case. Far more examples utilizing advanced capabilities of the polyhedral model, which have been omitted in this PM overview for brevity, can be found in the literature [22, 12]. Also, the utilized approach of first identifying the structure of the desired target code followed by designing the corresponding modifications is only one way of utilizing polyhedral transformations. As has been outlined above, the matrices describing a transformed code version may be computed utilizing mathematical optimization techniques, yielding results and complex code fragments that could only very difficulty obtained by sequences of conventional transformations.

5.4.2 Integration of Polyhedral Transformations

The integration of polyhedral transformations into the Insieme Compiler infrastructure is illustrated by Figure 5.1. In a first step, a targeted IR code fragment is converted into a polyhedral representation (PM) as it is done when analyzing codes. The representation is then transformed according to the desired PM based transformation and converted by the ClooG library into an AST like structure. This ClooG AST, combined with some information linking statements referenced by the polyhedral model to statements in the input IR structure, can be converted back into the Insieme IR. The transformed code is then returned as the result of the transformation.

5.5 The Transformation Framework

So far, utilities to build code transformations have been covered in this section. Each of those targets an IR node and replaces it by some sort of substitute. However, no structured approach for identifying those targets or to define the order of transformations to be applied has yet been introduced. Within this final section, such means for orchestrating transformations are covered.

5.5.1 Transformations and Connectors

The following definition constitute the transformation framework realized as part of the Insieme Compiler infrastructure.

The basic entity of the framework is the definition of a transformation.

Definition 5.16 (transformation)**.** A transformation is a function

$$t : \mathbb{IR} \to (\mathbb{IR} \cup \{\perp\})$$

converting a given IR node n into $t(n) \in \mathbb{IR}$ if the transformation can be validly applied on n or $t(n) = \perp$ otherwise. The set of all transformation functions is denoted by $\mathcal{T}$.

Such a transformation may be implemented based on plain node construction operations, the node mapper infrastructure, the pattern matcher, the polyhedral model, by composing other transformations or by utilizing any other available technique. Frequently, transformations are implemented generically such that they are exhibiting parameters. For instance, a loop-unrolling transformation may be implemented by offering the actual unroll factor as a parameter. Hence, loop-unrolling constitutes a whole family of transformations, referred to as a transformation type.

Definition 5.17 (transformation type)**.** A *transformation type* is a family of transformations parametrized by a parameter $p \in \mathcal{P}$ of some parameter space $\mathcal{P}$. Formally, a transformation type is a function

$$f : \mathcal{P} \to \mathcal{T}$$

obtaining a parameter $p \in \mathcal{P}$ the corresponding transformation $f(p) \in \mathcal{T}$.

Example 5.10 (transformation types)**.** For instance, the function

$$\text{unroll} : \mathbb{N} \to \mathcal{T}$$

defined by

$$\text{unroll}(n) = t_n$$

where $t_n : \mathbb{IR} \to (\mathbb{IR} \cup \{\perp\})$ is defined by

$$t_n(i) = \begin{cases} \text{n-times unrolled loop l} & \text{if } i = l \text{ is a for-loop} \\ \perp & \text{otherwise} \end{cases}$$

is the transformation type *unroll* and e.g. t_4 the transformation unrolling a loop with an unroll factor of 4.

Transformation types provide factory functions for transformations that can be applied on code fragments. The obtained transformations can then be orchestrated into a larger transformation utilizing transformation connectors. Some of those are based on node filters.

Definition 5.18 (node filter). Let $\mathbb{IR}_A \supset \mathbb{IR}$ be the set of node addresses (see Section 3.10.2). A *node filter* is a function

$$f : \mathbb{IR}_A \to \mathbb{B}$$

The set of all node filters is is denoted by $\mathcal{F}$.

Node filters are predicated on nodes. Those predicates may be realized by simple code inspection operations, by conducting analysis or matching patterns. As for transformations, filters are typically implemented generically by exposing parameters. A related family of filters is referred to as a filter type.

Definition 5.19 (node filter type). A *node filter type* is a family of node filters parametrized by a parameter $p \in \mathcal{P}$ of some parameters space $\mathcal{P}$. Formally, a node filter type is a function

$$f : \mathcal{P} \to \mathcal{F}$$

obtaining for each parameters $p \in \mathcal{P}$ the corresponding node filter $f(p) \in \mathcal{F}$.

Example 5.11 (node filters and types). A constant function

$$\text{accept} : \mathbb{IR}_A \to \mathbb{B}$$

defined by $\text{accept}(n) = \text{true}$ is a valid node filter accepting any node. A function

$$p : \Phi \to \mathcal{F}$$

defined by

$$p(\phi) = m_\phi$$

where $m_\phi : \mathbb{IR}_A \to \mathbb{B}$ is defined by

$$m_\phi(n) = (\text{match}(n, \phi) \neq \perp)$$

is another example of a pattern based node filter utilizing a tree pattern $\phi \in \Phi$ to identify nodes to be accepted.

Finally, the following set of transformation connectors can be defined based on the previous definitions.

Definition 5.20 (transformation connectors)**.** Let $t, t_1, \ldots, t_n \in \mathcal{T}$ be transformations and $f \in \mathcal{F}$ be a filter. Further, let $t_{[a/b]} \in \mathcal{T}$ denote a transformation replacing the node addressed by $b \in \mathbb{IR}_A$ by the value of $a \in \mathbb{IR}$, $a \sqsubseteq b$ denote the fact that $a \in \mathbb{IR}_A$ addresses a sub-structure of $b \in \mathbb{IR}_A$ and $a \succ b$ the fact that $a \in \mathbb{IR}_A$ addresses a node succeeding the node addressed by $b \in \mathbb{IR}_A$ in a *pre-order* iteration through the tree structure defined by the common root of a and b. Then

$$\begin{gathered}
\text{pipeline}(t_1, \ldots, t_k) \\
\text{try}(t_1, t_2) \\
\text{if}(f, t_1, t_2) \\
\text{forAll}(f, t) \\
\text{fixpoint}(t)
\end{gathered}$$

defined by

$$\begin{aligned}
\text{pipeline}()(n) &= n \\
\text{pipeline}(t_1, \ldots, t_k)(n) &= \text{pipeline}(t_2, \ldots, t_k)(t_1(n)) \\
\text{try}(t_1, t_2)(n) &= \begin{cases} t_1(n) & \text{if } t_1(n) \neq \bot \\ t_2(n) & \text{otherwise} \end{cases} \\
\text{if}(f, t_1, t_2)(n) &= \begin{cases} t_1(n) & \text{if } f(n) \\ t_2(n) & \text{otherwise} \end{cases} \\
\text{forAll}(f, t)(n) &= \text{pipeline}(t_{[t(n_1)/n_1]}, \ldots, t_{[t(n_k)/n_k]})(n) \\
&\quad\ \text{where } \{n_1, \ldots, n_k\} = \{x \in \mathbb{IR}_A \mid x \sqsubseteq n \wedge f(x)\} \\
&\quad\ \text{and } \forall_{1 \leq i < k} \,.\, n_i \succ n_{i+1} \\
\text{fixpoint}(t)(n) &= \begin{cases} n & \text{if } t(n) = n \\ \text{fixpoint}(t)(t(n)) & \text{otherwise} \end{cases}
\end{aligned}$$

are transformations as well.

Essentially, transformation connectors are higher-order transformation types utilizing other transformations as parameters. The *pipeline* operator creates sequences of transformations, the *try* and *if* operator apply transformations under certain conditions, the *forAll* operator apples a given transformation on selected sub-structures and the *fixpoint* operator repeatedly applies a given transformation until no more changes occur. For the latter case, it has to be ensured that such a fixpoint is reached eventually.

Combined with the remaining transformations the present connectors can be utilized to assemble arbitrary, parametrized transformation scripts.

Example 5.12 (innermost loop unrolling)**.** To demonstrate the expressiveness of the presented transformation connectors we will device a transformation unrolling all innermost for loops by a factor of 4. Let $id \in \mathcal{T}$ be the identity function, hence a transformation without any effect. The desired transformation is given by the term

$$\begin{aligned}&\text{forAll}(\\&\quad \text{p}(\text{innermost}(\text{forStmt}())),\\&\quad \text{try}(\text{unroll}(4), \text{id})\\&)\end{aligned}$$

where the pattern $\phi = \text{innermost}(\text{forStmt}()) \in \Phi$ is utilized to select the nodes to be targeted by the transformation $\text{try}(\text{unroll}(4), \text{id}) \in \mathcal{T}$ which attempts to apply the loop unrolling operation. Unlike the plain unrolling-transformation the resulting transformation may be applied on any node structure.

5.5.2 Implementation of Transformation Scripts

Like the pattern matcher and the rest of the Insieme System, the transformation framework is implemented based on C++ utilizing a functional programming style. Consequently, transformation scripts can be defined similar to the examples illustrated in the previous section by adapting it to C++ syntax.

The following code fragment outlines the construction of a transformation script for a *matrix-multiplication* kernel.

```
auto t = makePipeline(
    // cleanup
    sequentialize,

    // tile
    makeLoopStamping(tsC, 0u, 0u, 0u),
    makeLoopTiling({tsA, tsB, tsC}),

    // collapse tiled loop
    makeForAll(
      f::outermostLoops(),
      makeLoopCollapsing(1)
    ),

    // unroll innermost loop
    makeForAll(
      f::innermostLoops(),
      makeTry(
        makeTotalLoopUnrolling(),
        makeLoopUnrolling(4)
      )
    ),
        // parallelize selected loops
        t::makeForAll(f::pickLoop(0), parallelize),
        t::makeForAll(f::pickLoop(1), parallelize)
    );
```

The overall transformation is a sequence of transformations, constituted by the enclosing pipeline. The first step is a *sequentialize* operation removing any potential parallel loops from the targeted code fragment. The second step conducts 3-dimensional loop tiling based on the transformation parameters tsA, tsB and tsC corresponding to the tile sizes to be used. Before the tiling is applied, the *loop stamping transformation* splits the present loops such that each loop in the targeted loop nest exhibits an iteration range forming a multiple of the desired tile size and the remaining part is processed in a separate loop. This reduces the complexity of the boundary expressions of the tiled loops and may enable total loop-unrolling in the innermost loops. This unrolling is attempted in the following step before the last step parallelizes the two outermost loops.

The utilized filters pickLoop enable loops being addressed by their location in the code. The first loop is 0, the second 1. The first loop nested in loop 0 is loop 0-0, the second loop 0-1 and so forth.

The resulting parametrized transformation is specifically designed to be applied on a code fragment implementing a matrix multiplication. Other scripts for alternative code fragments may be equally composed.

5.6 Summary

The frameworks and utilities covered in this chapter provide the means for building and composing parametrized code transformations and transformation scripts. The range of currently supported techniques range from rudimentary approaches of implementing transformations by directly composing modified nodes (Section 5.2), over node mapper (Section 5.2.1) and novel pattern-matching based techniques developed as part of this thesis (Section 5.3), to sophisticated polyhedral transformations (Section 5.4). The application of the resulting elementary, local transformation operations can be orchestrated utilizing the high-level primitives of the transformation framework to form transformation scripts (Section 5.5). However, the problem of devising (suitable) scripts and/or optimal parameters for the involved transformation types is known as *auto-tuning* of program codes and among the topics of the next chapter – which also includes a variety of example applications utilizing the transformation infrastructure presented in this chapter.

Chapter 6

Applications

All compiler optimizations require three components: analyses determining their validity, transformations conducting the actual code manipulations and a third, decision making component determining where to apply transformations and whether they are actually profitable. The first two components have been covered in the last two chapters, the final is to be covered in this chapter.

This chapter discusses a few example applications and research results based on the Insieme infrastructure developed in the context of this thesis. Each section summarizes a different application presented in a different publication utilizing the Insieme infrastructure for its implementation. The first covers a multi-objective auto-tuning framework for parallel applications [49], the second an automated scheduling approach for parallel loops [97] and the third a hybrid compiler/runtime optimization improving the execution of nested parallel codes [99, 100]. The chapter concludes by a short section enumerating a list of additional work conducted by third parties, yet utilizing the infrastructure developed by this thesis.

6.1 Contributions

The major contributions of this chapter are:

- the demonstration of the utilization of the Insieme infrastructure for conducting research on the automated tuning and coordination of parallel programs
- the development of a novel system for the multi-objective tuning of (parallel) programs (Section 6.2)
- the development of a novel loop-scheduling solution based on a runtime system component utilizing compiler deduced properties characterizing the scheduled loops (Section 6.3)

- the development of a novel, automated task-granularity control mechanism reducing the overhead and coordination effort of task parallel applications based on a combination of compiler and runtime system based techniques (Section 6.4)

In the context of this thesis, this chapter corroborates the thesis's hypothesis by demonstrating a variety of applications utilizing the widened influence of a novel, high-level, parallelism aware source-to-source compiler for automatizing the load management, tuning and coordination of parallel programs.

6.2 Multi-Objective Auto-Tuning

Efficient parallelization and optimization of programs for modern parallel architectures is a time-consuming and error-prone task that requires numerous iterations of code transformations, tuning and performance analysis which in many cases have to be redone for each different architecture.

Auto-tuning has been extensively studied in recent years to largely automate the process of tuning codes for (parallel) computers to realize portable performance [64, 111, 109, 23, 33]. The basic idea is to automatically find an effective set of transformations with proper parameter settings (e.g. tile sizes, loop ordering and unrolling factors) for individual code regions. However, while a carefully selected and tuned transformation sequence might be beneficial for one objective, the same may have adverse consequences on others [91]. Many successful practical auto-tuning solutions focus on specific applications [111, 109]. For more generic, compiler-based approaches, however, the prohibitively large and complex optimization problem of selecting, customizing and ordering transformations to obtain an optimal variant of a user's input code is still among the most fundamental open issues in compiler research [102, 86, 11, 91, 54].

Existing auto-tuning compilers often try to tackle this problem by (1) offline searching a subset of the parameter space [102] based on domain specific constraints and heuristics or analytical performance models [91], (2) online tuning of program parameters (e.g. tile sizes) [11], or (3) dynamic code generation and replacement [103]. One factor common to most of these methods and systems, particularly in the field of parallel computing, is that they focus exclusively on a single optimization objective such as execution time, memory behavior or resource consumption. Only little support exists for code optimizers that deal with trade-offs between multiple, conflicting goals.

Based on the Insieme infrastructure, a novel multi-objective auto-tuning framework for parallel codes has been introduced [49]. It consists of a compiler component featuring a multi-objective optimizer and a runtime system. The multi-objective optimizer derives a set of non-dominated solutions, each of them expressing a trade-off among the different conflicting objectives.

This set is commonly known as *Pareto set* in the field of multi-objective optimization research [21].

To make effective use of the resulting Pareto set of optimal solutions, each of them has to be made available at runtime. This is achieved by having the compiler generate a set of code versions per region, each corresponding to one specific solution. The runtime system then exploits the trade-off among the different objectives by selecting a specific solution (code version) for each region, based on context-specific criteria.

The approach is generic and can be applied to arbitrary transformations and parameter settings. Its effectiveness is demonstrated by exploring the trade-off between execution time and efficiency when tuning tile-sizes and the numbers of involved threads in shared memory parallel applications. This full section is a summary of previously published work [49].

The major contributions of the author of this thesis to this Insieme based application are – besides the development of the underlying infrastructure – the design and implementation of the multi-objective optimization infrastructure covered in Section 6.2.2, including the required static and dynamic analyses, code transformations and the backend support, as well as the implementation of the random and brute force optimizer and the conduction and evaluation of the involved experiments.

6.2.1 Motivation

Two separate but related observations motivate the design of the multi-objective auto-tuner. Firstly, it is well-known that for many computational problems, strong parallel scaling can not be achieved. Beyond some threshold – the exact value of which is problem, architecture and implementation dependent – increasing the number of threads (cores) will no longer sufficiently decrease computation time, which results in an inefficient use of the available resources. The resulting trade-off between efficiency and speedup motivates the use of multi-objective optimization techniques in compiler research. Figure 6.1 illustrates this trade-off on one of the parallel computers and benchmarks used in our evaluation (see Section 6.2.3 for details).

A second observation, which motivated the use of multi-versioning for our compilation approach, is that different numbers of threads may require specific transformation parameter values or even distinct transformation sequences for optimal performance. A potential reason for this is that the effective capacity of shared cache levels exploitable by individual threads depends on the number of concurrent threads running on the same chip. In Figure 6.2 this behavior is illustrated for different tile sizes in the case of three-dimensional tiling for matrix multiplication (IJK loop ordering). The images show the relative execution time of tile size combinations for the i and j loop, keeping the tile size for the k loop fixed. Darker areas represent faster tile size combinations. It can be seen that the selection of optimal tile

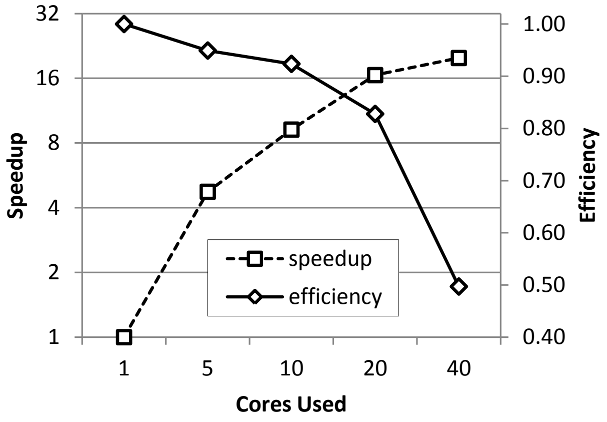


Figure 6.1: Efficiency and speedup trade-off in a matrix multiplication kernel for a varying number of cores.

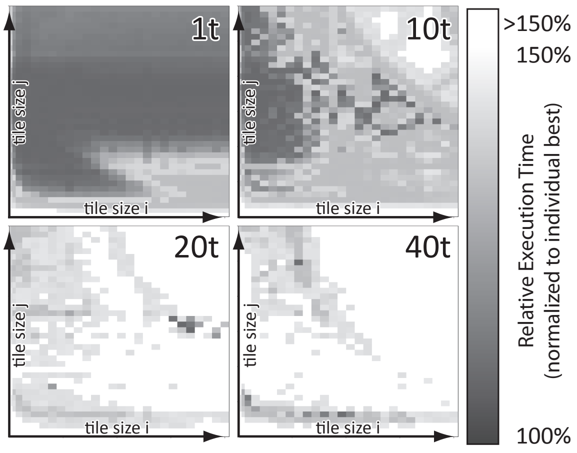


Figure 6.2: Relative execution time of tiled matrix multiplication with different tile size selections, for 1, 10, 20 and 40 threads.

sizes depends on the amount of threads used in the computation – a pattern that has also been observed by Shirako et al. [91].

These preliminary results demonstrate that, in order to offer the highest possible performance at any desired efficiency level – and thus thread count – an optimizing multi-objective compiler has to include some adaptive mechanism to specialize each tuned parallel code section for a given number of threads.

6.2.2 Method

Our method, which is based on a combination of compiler and runtime techniques, is described in this section. The first part gives an overview of

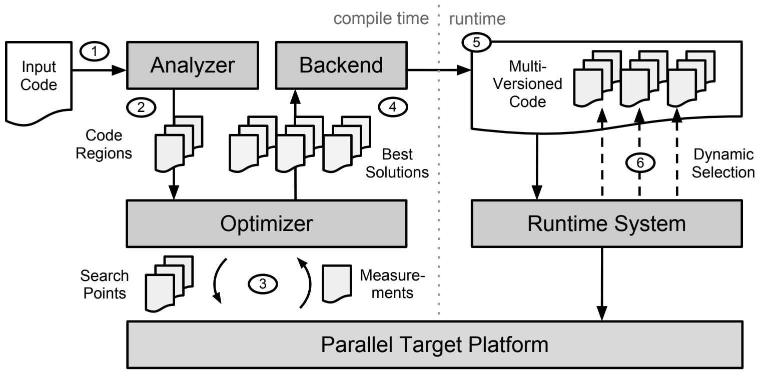


Figure 6.3: Overview of our multi-objective optimization infrastructure.

the general architecture of our approach based on the Insieme infrastructure, whereas the second part provides a brief overview on the utilized multi-objective optimization algorithm.

Architecture Overview

Figure 6.3 illustrates the overall architecture of our solution, highlighting the four main components: the code analyzer, the multi-objective optimizer, the multi-versioning backend and the runtime system. The labels (1-6) in Figure 6.3 follow the processing of a program within our framework. An input code will be loaded by the compiler (1), analyzed and decomposed into regions to be tuned by the optimizer.

For each region, the analyzer determines a set of *transformation skeletons* (=parametrized transformation scripts, see Section 5.5) which describe generic sequences of code transformations using unbound *parameters* for tunable properties (e.g. tile sizes, unrolling factors or number of threads).

The regions, together with their associated transformation skeletons and some (optional) parameter constraints, are passed on to the *optimizer* specifically devised for the auto-tuning system (2). At this point, the optimizer conducts auto-tuning by iteratively selecting sets of *configurations* for each of the regions to be evaluated (executed) on the target system (3). Each configuration corresponds to an instantiation of a transformation skeleton's parameters. During the evaluation, a single execution of the resulting program is sufficient to obtain measurements for all simultaneously tuned regions. For this purpose the dynamic analysis infrastructure covered by Section 4.6 is utilized. To further reduce the optimization time, multiple independent configurations are generated, compiled and if possible evaluated in parallel on distinct instances of the targeted platform.

In the end, the optimizer derives a Pareto set for each of the selected regions (4). The backend then generates, for each code region, a set of specialized *code versions* – one for each solution in the obtained Pareto set (5). For this purpose the multi-versioning capabilities of work items are utilized in the runtime system (see Section 2.4.1). Additionally, each code version is annotated with meta-information to be used by the runtime system, including details regarding the represented trade-off.

Finally, during execution of the resulting code region, the runtime system dynamically selects among the available code versions (6). The actual strategy used to do so remains application specific. The decision might be forwarded to the user. Alternatively, system wide performance settings may be considered. In more sophisticated scenarios, dynamic or static task schedulers could be extended to exploit this additional flexibility to improve their own (potentially multi-objective) quality of service. However, the ultimate method to utilize the newly gained opportunity of dynamically customizing non-functional attributes is beyond the scope of this work and left for future research.

The Static Optimizer Algorithm

The role of the static optimizer is to tune the transformation skeletons provided for the selected regions by computing a set of good configurations (solutions) for each of them.

Multi-Objective Optimization A multi-objective optimization problem is characterized by an *objective function*

$$f : C \to \mathbb{R}^m$$

where C is the set of all valid configurations and $m \geq 2$ the number of objectives. A configuration $c_1 \in C$ dominates a configuration $c_2 \in C$, if c_1 provides better results than c_2 for all objectives. More accurately, whenever

$$\forall_{1 \leq i \leq m} \, . \, f_i(c_1) \leq f_i(c_2)$$

and

$$\exists_{1 \leq i \leq m} \, . \, f_i(c_1) < f_1(c_2)$$

is satisfied. On the contrary, two configurations are non-dominated if neither is dominating the other. A set of non-dominated configurations is called *Pareto set*. A Pareto set is said to be *optimal* when no configuration $c' \in C$ dominates any configuration in it. The goal of a multi-objective optimization algorithm is to find a Pareto set $S \subseteq C$ as close as possible to the optimal Pareto set.

Within the static optimizer, the problem of tuning a region is mapped to an instance of a multi-objective optimization problem and solved using a generic solver. Therefore, the search space C is derived from the parameters exhibited by the associated skeletons. Within each configuration all tuning options, including the skeleton to be selected, potential flags enabling optional parts of the transformation skeleton, unrolling factors, tile sizes and thread count specifications are modeled uniformly. Furthermore, each configuration comprises all the information required to convert a code region into a code variant which can be evaluated by executing it on the target system. This process is performed by the implementation of the objective function f, which executes the resulting version and collects measurements to quantify the individual objectives (e.g. execution time, resource usage, energy consumption, etc.).

Multi-Objective Optimization Techniques Usually, the cardinality of the set C is prohibitively large, rendering it impossible to perform an exhaustive search evaluating all the configurations. In order to deal with this problem, approximation techniques are employed. The challenge is to compute solutions that are close to the optimal Pareto set by evaluating only a minimal number of configurations.

Traditionally, approximation techniques from the field of operational research like Nelder-Mead, Simplex or Genetic Algorithms (GA) [66, 103, 54] have been applied within optimizing compilers. Although these techniques only evaluate a small fraction of the overall search space, the number of steps is still too large to represent a viable option to be used within a compiler. In order to overcome this obstacle, several complementary methods for additionally reducing the search space have been proposed [91]. The effectiveness of most of these methods relies on analytical prediction models that are largely domain specific. Furthermore, these approaches only focus on a single objective. It has not yet been demonstrated whether these techniques can be extended to multi-objective optimization problems.

In our approach, we utilize a novel multi-objective optimization algorithm to be used within our iterative compiler framework. We refer to this algorithm as *RS-GDE3*. It combines an approximation technique from the class of Differential Evolution (DE) algorithms [95] with a reduction mechanism based on Rough Set theory [76]. Unlike other reduction mechanisms, our selected concept does not depend on any domain-specific knowledge. To reduce the search space, the Rough Set based approach requires only a small number of evaluated configurations. Consequently, our approach does not depend on any analytical models or heuristics to reduce the search space – making it de facto independent of the actual interpretation of the tuned parameters. The full details on the optimization algorithm and the utilization of its sub-components are covered in the corresponding literature [49].

In every iteration, RS-GDE3 produces a set of new configurations to be evaluated on a target architecture – one for each element within the current population. We have configured RS-GDE3 for our work to stop

System	Sockets/ Cores	Cache L1d/i	Cache L2	Cache L3	Setup Kernel	Setup GCC
Westmere	4/40	32K/32K	256K	30M	2.6.32	4.5.3
Barcelona	8/32	64K/64K	512K	2M	2.6.18	4.5.3

Table 6.1: Hardware platforms for experimental evaluation.

```
1: for $i = 1 \dots N$ do
2:     for $j = 1 \dots N$ do
3:         for $k = 1 \dots N$ do
4:             $C[i][j] = C[i][j] + A[i][k] * B[k][j]$
5:         end for
6:     end for
7: end for
```

Figure 6.4: Simple MM kernel using IJK loop ordering.

iterating when the solutions do not improve for three consecutive iterations. A particular advantage of RS-GDE3 is that configurations can be evaluated simultaneously to reduce the search time, a property exploited by our autotuner by evaluating configurations in parallel.

6.2.3 Results

Experimental Setup

To demonstrate the capabilities of our system we used two parallel computing systems. The first, an Intel-based system, incorporates 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. We refer to this system as *Westmere*. The second system is an AMD server, named *Barcelona*, which is based on 8 Opteron 8356 processors, each contributing 4 cores. Table 6.1 summarizes the configuration of these systems. Note that L1 and L2 are per-core private caches, whereas L3 is shared among the cores of each CPU.

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. For our experiments we observed that the hyper-threading support on the Intel architecture did not provide any extra benefit – neither in terms of speed nor efficiency. Therefore, we are skipping the corresponding results for brevity.

A Detailed Example

To provide a detailed analysis of our approach we have selected the widely known and investigated *Matrix Multiplication* (mm) kernel as shown within

Figure 6.4. We are applying loop tiling on all three loop levels, thereby creating a three-dimensional parameter space. This transformation is followed by collapsing the two outermost tiling loops (corresponding to the i and j loops) to increase the number of iterations before parallelizing the resulting outermost loop.

Unlike in the sequential or the non-tiled case, using the (initially) more cache friendly IKJ loop ordering as a starting point is not an option due to a inherent load balancing issue introduced by doing so. When tiling the IKJ variant of an mm-kernel using (necessarily) large tile sizes $t = (t_i, t_k, t_j)$ to obtain a good last level cache utilization, the number of loop iterations is limited by $\lceil N/t_i \rceil$ due to tiling. The larger t_i the less iterations can be distributed among the threads processing the loop. Additionally, each iteration computes larger sections of the resulting matrix. As a consequence load balancing can degrade, which may seriously limit the scalability for an increasing number of cores. To mitigate this effect, loop collapsing can be applied before parallelizing the outermost loop, thereby effectively reducing the work-load associated with an individual iteration. However, in the IKJ case collapsing the tiling-loops of i and k prohibits the intended subsequent parallelization. Since this limitation does not occur when starting with the IJK loop ordering, we have selected the IJK loop ordering for our evaluation.

Tiling of Parallel Loops We first want to investigate the influence of the number of threads used for processing a parallel loop nest on the optimal tile size for this loop. Therefore, we conducted an extensive search within a necessarily restricted search space on two different architectures. For a problem size of $N = 1400$, mm-kernel variations using more than 14.000 tiling configurations $t = (t_i, t_j, t_k) \in [1..700]^3$ have been generated, compiled and evaluated on our target platforms for different thread numbers. On Westmere we performed our evaluation using $1, 5, 10, 20$ and 40 cores while on Barcelona configurations involving $1, 2, 4, 8, 16$ and 32 cores have been investigated. We refer to this kind of extensive search as the *brute force* method. A subset of the results has been illustrated in Figure 6.2.

Each of the resulting configurations has been evaluated multiple times and the median of the collected execution times was used for comparison. Table 6.2 lists the best tile sizes found for each specific number of employed threads on our target platforms. For instance, the best configuration for 10 cores found on Westmere is $(t_i, t_j, t_k) = (32, 288, 9)$. The table also includes the relative performance loss when running an optimal configuration obtained for one number of threads using a different number of cores. For instance, using 40 threads to run the mm kernel version tuned for 10 threads takes 11% longer than the variant tuned specifically for 40 threads.

As can be observed in Table 6.2, on both architectures the performance impact of using tiling parameters tuned for a non-matching number of

Westmere Architecture

	opt. Tile Sizes			Perf. Loss over Best for # of Cores in %					
Nr. of Cores	t_i	t_j	t_k	1	5	10	20	40	Avg.
1 core	96	128	8	-	1.2	5.4	11.3	15.1	8.3
5 cores	32	304	9	1.0	-	1.5	4.5	0.3	1.8
10 cores	32	288	9	0.9	0.1	-	4.3	11.0	4.1
20 cores	32	192	12	2.3	2.2	2.5	-	10.4	4.4
40 cores	32	208	12	1.7	0.4	3.2	6.8	-	3.0
GCC -O3	-	-	-	605.3	593.3	559.7	502.6	422.8	536.8

Barcelona Architecture

	opt. Tile Sizes			Perf. Loss over Best for # of Cores in %						
Nr. of Cores	t_i	t_j	t_k	1	2	4	8	16	32	Avg.
1 core	96	480	5	-	7.3	7.3	10.7	6.0	18.0	9.9
2 cores	80	496	5	0.6	-	4.5	5.1	14.9	17.1	8.4
4 cores	128	352	7	1.4	4.3	-	12.1	4.2	19.3	8.3
8 cores	176	304	7	6.7	3.1	5.5	-	23.2	30.1	13.7
16 cores	176	352	8	7.4	6.6	8.0	2.1	-	2.9	5.4
32 cores	144	240	8	5.9	5.3	9.8	22.0	13.5	-	11.3
GCC -O3	-	-	-	3948.7	3732.9	3619.9	3886.5	3486.5	4628.0	3883.8

Table 6.2: Optimal Tiling Parameters for Different Number of Threads and Architectures.

threads can be quite severe. For instance, if we apply the best tile size configuration found for a single thread to a matrix multiply execution with all available cores on our target architectures, then the resulting performance degradation amounts to 15.1% and 18%, respectively, compared to the best found tile size for the largest number of available threads. On our Barcelona system, this performance impact even reaches 30.1% when using the configuration tuned for 8 cores for a mm run involving 32 cores. Also, when using configurations obtained for a large number of cores in scenarios involving fewer threads, a slowdown can be observed. Using the best 32-core configuration on the Barcelona system with less threads results in performance loss between 5.3% and 22% compared to the individual best configurations.

Configurations representing the optimum for a single number of cores are on average between 1.8% and 8.3% (column Avg. in Table 6.2) slower than individually tuned solutions on the Westmere architecture. On the Barcelona architecture this range varies between 5.4% and 13.7%. Finally, the comparison with the GCC -O3 baseline demonstrates the well known, enormous potential of tiling in general as well as the high quality of the individually tuned solutions.

Trade-off between Speedup and Efficiency In addition to the impact on optimal tiling parameters, the collected data also enables us to investigate the trade-off between speedup and efficiency when tuning the mm-kernel for multiple objectives.

Let t_s be the execution time of the fastest (tiled) sequential code version and $t_p(x)$ its parallel counterpart using x threads. The speedup of a code being executed using x threads is usually defined by

$$s(x) = \frac{t_s}{t_p(x)}$$

and its efficiency by

$$e(x) = \frac{s(x)}{x} = \frac{t_s}{x \cdot t_p(x)}$$

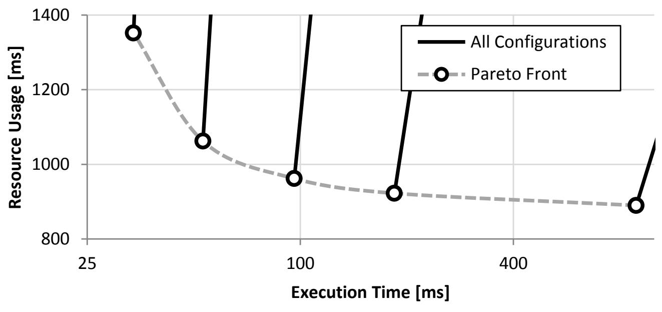
Unfortunately, both definitions depend on the fastest sequential execution time, which in general remains unknown. However, t_s is a constant which can be omitted when comparing the quality of different versions of the same region. Hence, instead of comparing the speedup $s(x)$ we can use the execution time $t_p(x)$ when trying to derive the fastest solution - just as the total *resource usage*

$$r(x) = x \cdot t_p(x)$$

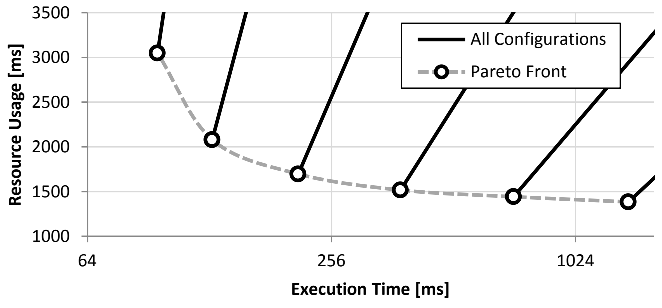
can be used as an equivalent, yet obtainable substitute for $e(x)$ as part of our optimizer. However, in general we strive to maximize $e(x)$ whereas $r(x)$ needs to be minimized.

When plotting execution time vs. the resource usage of all the configurations evaluated using the brute force search mechanism, the pattern illustrated within Figure 6.5 is produced. Due to the correlation between the execution time $t_p(x)$ and the resource usage $r(x) = x \cdot t_p(x)$, all points using the same number of threads are located on a line. Furthermore, due to the large number of points evaluated, the individual lines are densely populated within a certain range. In every line, there is a single point exhibiting the shortest execution time and thus also the least resource requirements. This point dominates all other solutions within the same line and corresponds to the best solutions listed in Table 6.2.

The globally non-dominated tips of the lines are the desired configurations to be obtained using our static multi-objective optimizer. Together, they form the Pareto front of the multi-objective optimization problem for speedup and efficiency. Unfortunately, finding these optimal solutions requires searching the optimal tiling parameters for individual thread counts in multiple vast, three-dimensional search spaces. For our evaluation we only considered 5 respectively 6 out of 40 respectively 32 options for the number of cores – all of them contributing one point to the Pareto front, which might not always be the case. Each additional number of cores adds another line which may add one extra point on the Pareto front. This problem definition automatically eliminates configurations using too many cores for non-scaling codes. For these cases, the execution time will increase for larger number of cores and thus their corresponding configurations will not be part of the



(a) Westmere Architecture



(b) Barcelona Architecture

Figure 6.5: Execution time and resource usage for different configurations evaluated based on brute force.

Pareto front. The same happens to solutions using an inadequate number of cores. For instance, if a region is only fully utilizing the available threads if their number is a power-of-two, all other thread numbers will result in solutions dominated by a configuration using fewer cores due to their bad resource utilization. Those will also be automatically discarded.

Table 6.3 lists the properties of the optimal points within the Pareto front and provides details about the trade-off between speedup and efficiency for each of these points.

The Optimizers' Solutions In a final step, we processed the mm-kernel using our static optimizer as well as a random search strategy for comparison. Unlike within the brute force case, where artificial restrictions had to be added to facilitate its completion within a reasonable time frame, only few search-space restrictions need to be defined for our optimizer. The upper boundary for tile sizes has been set to $N/2$, since larger tile sizes clearly have little potential to dominate smaller tile sizes. Further, the upper boundary for the number of threads was set according to the target machine. Both restrictions could easily be extracted statically from the targeted region and

Westmere Architecture

Cores	Speedup	Efficiency	Relative Time	Relative Resources
1	1.00000	1.00000	100%	100%
5	4.82873	0.96575	21%	104%
10	9.26091	0.92609	11%	108%
20	16.77778	0.83889	6%	119%
40	26.35799	0.65895	4%	152%

Barcelona Architecture

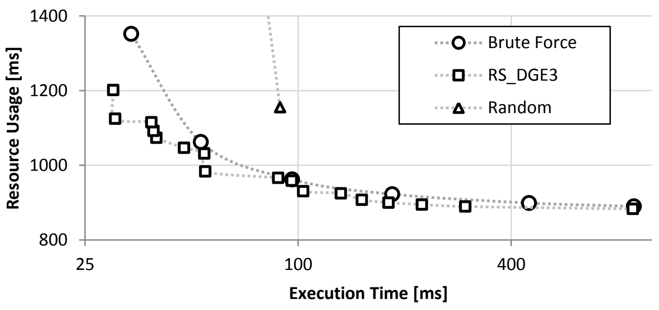
Cores	Speedup	Efficiency	Relative Time	Relative Resources
1	1.00000	1.00000	100%	100%
2	1.92067	0.96033	52%	104%
4	3.65286	0.91322	27%	110%
8	6.53208	0.81651	15%	123%
16	10.65231	0.66577	9%	150%
32	14.53095	0.45409	7%	220%

Table 6.3: Impact of Number of Threads on Speedup and Efficiency.

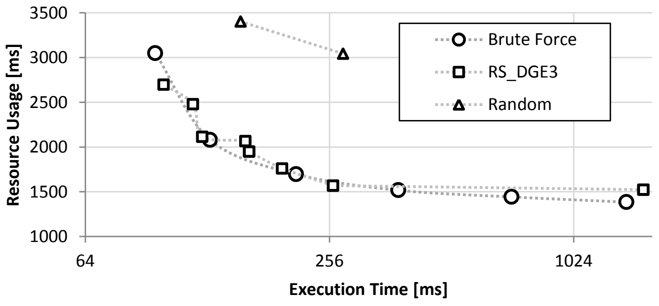
platform. No constrains regarding the granularity of the potential configurations have been defined.

Figure 6.6 compares the Pareto front obtained by our optimizer with the ones obtained after exploring the search space with the brute force mechanism and a random search in both architectures. The implemented random search generates random configurations, evaluates them and returns those which are non-dominated. In the case of the Westmere architecture, we observe that the configurations generated by our optimizer are better (lower execution time and better utilization of resources) than the solutions generated by the brute force approach. For this specific architecture, our algorithm produces solutions that are up to 13% faster. In the case of the Barcelona architecture, our solutions are close to the brute force results. In both architectures, random search using an equal number of evaluations as our method is very far off the quality achieved by the other techniques.

To systematically compare different optimization strategies we are using three derived metrics. Let S be the set of solutions obtained by an algorithm. The first, simplest metric is $|S|$, the number of points within the set. Since a larger number of solutions offers a higher flexibility for the dynamic decision-making, this quantitative metric has been included. Additionally, to judge the quality of a solution set S, its *hypervolume* $V(S) \in [0 \ldots 1]$ [116] is employed as a metric. It computes the normalized volume (in the bi-objective case the area) behind a front. The larger $V(S)$, the closer the front could be pushed toward the hypothetical ideal $(0, 0)$ point. Hence, the hypervolume provides a metric for the quality of the individually obtained



(a) Westmere Architecture



(b) Barcelona Architecture

Figure 6.6: Pareto fronts obtained using different algorithms.

solutions, ranging from 0 (worst solutions) to 1 (unattainable optimal solutions). Finally, we compare the number of points E evaluated for obtaining a solution set, representing a metric for the time required to apply the corresponding method. Unlike the other two metrics, E does not describe the quality of the obtained solution but provides an indicator of the efficiency of the algorithm itself.

Since two out of three of the covered search strategies are stochastic algorithms, they produce different results in different runs. Hence, the results of a single run are not sufficient for an objective comparison. Thus, results collected from repeated executions have to be aggregated and compared. In our evaluation we use the arithmetic means $\overline{E}$, $\overline{|S|}$ and $\overline{V(S)}$ derived by running the optimizer 5 times as a directly comparable substitute.

The results of this comparison for the mm-kernel are listed in Table 6.6 together with the results for the kernels investigated in the following subsection. The metrics reflect the observation made in Figure 6.6. For the Westmere architecture, our algorithm obtains 9.4 solutions on average, which are exceeding the quality of the solutions obtained using the brute force approach although evaluating fewer than 1.1% of the points within the search space. For the Barcelona system, 10.4 solutions with a slightly weaker per-

Kernel	Problem Size	Computation	Memory
mm	1400^2	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
dsyrk	1400^2	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
jacobi-2d	10000^2	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
3d-stencil	600^3	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$
n-body	500000	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$

Table 6.4: Kernel Characteristis

formance are obtained on average by only evaluating less than 0.9% of the points the brute force approach touches. In both cases, our RS-DGE3 algorithm is vastly outperforming the much simpler random search strategy covering a comparable number of points.

Additional Kernels

Finally, we would like to demonstrate the general applicability of our optimization scheme. To this end, four additional kernel codes, each exhibiting different computation and memory usage characteristics, have been selected and evaluated.

The set consists of one additional BLAS-3 linear algebra kernel (*dsyrk*, computing $B = A*A^T+B$), two stencil codes (*jacobi-2d* and a generic 3x3x3 *3d-stencil*) and a naive implementation of an *n-body* simulation. Except for the *mm* and *dsyrk* kernels, all of them exhibit distinct computation / memory complexities as listed in Table 6.4 and hence considerably different memory reuse and access patterns. Also, although identically categorized in terms of complexity, the memory access patterns of *mm* and *dsyrk* are very different since the (on-the-fly) transposition of A eliminates the unaligned matrix access conducted within the *mm* kernel.

Table 6.5 summarizes the impact of thread-specific optimization when applying brute force on the selected kernels. Each row corresponds to the last column included in Table 6.2, representing the average performance loss when applying the ideal tile size for some particular number of threads across all other thread counts. Additionally, the overall average loss (*avg*) and the maximum loss incurred when only optimizing for serial execution (*1tmax*) are included.

In the *jacobi-2d* kernel, the average performance loss across all thread numbers is 11.8% on Westmere, while it goes up to 28.7% on Barcelona. Conversely, *3d-stencil* averages 24.6% on Westmere and only 14.7% on Barcelona, demonstrating that the impact of choosing distinct per-thread-count tile parameters varies across both hardware platforms and software applications. This is even more pronounced in the *n-body* kernel, where our chosen problem size fits entirely in the cache on Westmere, resulting in almost no variation, while the performance loss is extremely significant on

Westmere Architecture

	% Avg. Perf. Loss of Best Params for							
Kernel	1t	5t	10t	20t	40t		avg	1tmax
mm	8.3	1.8	4.1	4.4	3.0		4.3	15.1
dsyrk	2.9	6.4	6.5	2.5	2.4		4.1	6.5
jacobi-2d	14.0	9.5	7.6	14.2	13.7		11.8	23.6
3d-stencil	8.6	17.1	75.1	6.6	15.8		24.6	26.7
n-body	0.7	0.3	0.5	0.4	0.5		0.5	1.5

Barcelona Architecture

	% Avg. Perf. Loss of Best Params for							
Kernel	1t	2t	4t	8t	16t	32t	avg	1tmax
mm	9.9	8.4	8.3	13.7	5.4	11.3	9.5	17.9
dsyrk	7.6	9.8	4.6	6.1	2.1	3.3	6.7	14.2
jacobi-2d	38.6	28.9	17.7	17.2	50.3	19.5	28.7	89.2
3d-stencil	70.8	5.9	2.4	3.8	2.4	3.0	14.7	119.4
n-body	112.1	116.2	110.4	29.7	28.4	27.6	70.7	293.0

Table 6.5: Brute Force Result Summary

Barcelona, averaging 70.7% due to its limited 2 MB L3 cache. We included the *1tmax* column to point out the potentially large losses incurred when optimizing for serial performance and using the same results for parallel execution. Particularly on the Barcelona system, execution times can increase by up to a factor of 4 (293.0% loss) with this approach, and losses are above 89.2% for *jacobi-2d*, *3d-stencil* and *n-body*.

Finally, we applied our optimizer to the range of investigated kernels and compared the results with the ones obtained by the brute force evaluation and a random search. The results of this comparison are listed within Table 6.6.

Several conclusions can be extracted from these results. First: our optimizer always computed more configurations than both the brute force algorithm and the random search. Second: the number of configurations evaluated by our optimizer were between 99% and 90% lower than the evaluations required by brute force. Third: the hypervolumes of the Pareto sets computed by our technique are comparable to the volumes of the sets obtained by brute force. In the case of the Westmere architecture, sets exceeding the quality of the brute force solutions could be obtained for all the analyzed kernels. In the case of the Barcelona architecture our approach computed Pareto sets of higher quality for the *jacobi-2d* problem. Only the *n-body* problem on Westmere posed some difficulties to our technique. These difficulties could be related to the shape of the search space of that particular application. Analyzing and resolving this issue in detail will be the focal

Westmere Architecture

	Brute Force			Random			RS-DGE3		
Benchmark	E	$\|S\|$	$V(S)$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$
mm	71290	5	0.84	780	2.0	0.03	724	9.4	0.88
dsyrk	71290	5	0.75	1200	4.4	0.00	1186	11.2	0.81
jacobi-2d	23805	4	0.69	825	10.8	0.73	1027	21.2	0.83
3d-stencil	10580	5	0.77	1000	8.0	0.52	852	21.4	0.86
n-body	26136	5	0.86	1350	3.2	0.65	1334	28.6	0.95

Barcelona Architecture

	Brute Force			Random			RS-DGE3		
Benchmark	E	$\|S\|$	$V(S)$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$	$\overline{E}$	$\overline{\|S\|}$	$\overline{V(S)}$
mm	85548	6	0.83	800	2.0	0.01	740	10.4	0.76
dsyrk	85548	6	0.87	1200	3.6	0.00	1152	11.0	0.78
jacobi-2d	28566	6	0.73	675	10.8	0.81	674	17.2	0.88
3d-stencil	12696	6	0.87	850	9.4	0.78	822	17.0	0.85
n-body	21780	6	0.78	1100	3.6	0.51	1063.6	25.6	0.68

Table 6.6: Pareto fronts obtained using different optimization algorithms

point of future work. For the sake of completeness, it should be noted that the results obtained by the optimizer clearly outperform the results of the random search in all cases.

6.2.4 Conclusion

Based on the Insieme infrastructure we established a compiler infrastructure capable of optimizing programs according to multiple, potentially conflicting objectives simultaneously. By exposing a set of individually tuned versions of a single code region to the runtime system, trade-off decisions can be deferred until the actual execution. This way, our system enables application developers to tune their codes automatically for multiple objectives without the requirement of fixing any priorities. This decision is left to the end user.

We demonstrated the capabilities of our framework, and in particular of our optimization algorithm, by tuning tiling parameters and thread-counts for five different parallel codes across two hardware platforms. Our search algorithm generally achieves results on par with or better than a brute force search while using 90% to 99% fewer evaluations. Additionally, our experiments provide evidence for the importance of considering the number of involved threads when tuning tile sizes. Failing to do so can decrease performance by up to a factor of 4.

6.3 Automated Loop Scheduling

OpenMP is one of the most widely used languages for programming parallel shared memory systems. The predominant paradigm to utilize OpenMP for increasing the performance of applications is to parallelize loops. Thereby, the runtime environment, controlled by the user, has to determine how to distributed the various potentially concurrent loop iterations among the available hardware cores. The corresponding decision is a trait-off between introduced management overhead and load balancing issues and may therefore have significant impact on the overall performance of a program. Consequently, this *loop scheduling* problem has been of interest since the standard's inception.

Based on the Insieme infrastructure, a fully automated loop scheduling system has been devised that does not depend on user decisions. The basic idea is to utilize static analyses within the compiler to characterize loop bodies to enable more informed scheduling decisions during the runtime. Furthermore, the runtime system continuously monitors the system environment and program state and considers those when conducting scheduling decisions. The resulting solution is applicable to existing OpenMP programs without any code-level changes.

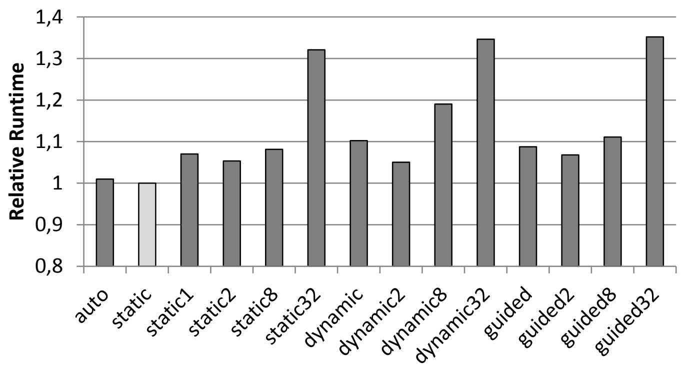
The content of this section is a summary of previously published work by Thoman, Jordan, Pellegrini and Fahringer [97] and derived work [98]. Besides the development of the underlying infrastructure, the major contributions of the author of this thesis to this application are the static analyses utilized for characterizing loop bodies as well as the design and implementation of the mechanism forwarding information from the compiler to the runtime system. All of those are covered in detail in Section 6.3.2.

6.3.1 Motivation

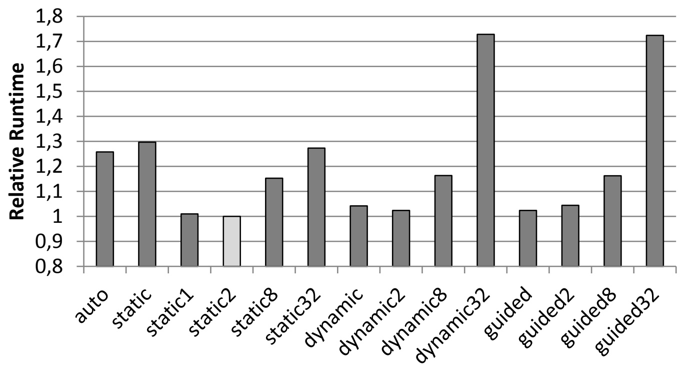
To motivate our design of a unified compiler/runtime approach to loop scheduling some initial experiments using simple OpenMP kernels in a variety of settings shall be presented. They also demonstrate the importance of load awareness. For a complete description of the experimental setup and hardware see Section 6.3.3. In all figures the relative execution time normalized to the best performing configuration is shown.

Figure 6.7 illustrates results for two kernels, dense matrix multiplication with full and triangular matrices, using a variety of standard OpenMP loop scheduling policies. Clearly, the ideal loop schedule depends on the characteristics of the program. The dense matrix multiplication exposes an equal amount of work within each iteration of the parallel loop while for the triangular matrix, the effort per iteration depends on the iterator value. We say that the dense matrix multiplication has a *flat work profile* while the work profile for the triangular matrix is *slanted.*

In the next experiment the impact of the problem size on the ideal loop schedule is investigated. In Figure 6.8 we see that with small problem sizes, the negative performance impact of scheduling policies with a runtime component (dynamic, guided) increases, most likely due to thread scheduling overhead. Also, the increase in workload per chunk mitigates the slightly



(a) Dense Matrix Multiplication



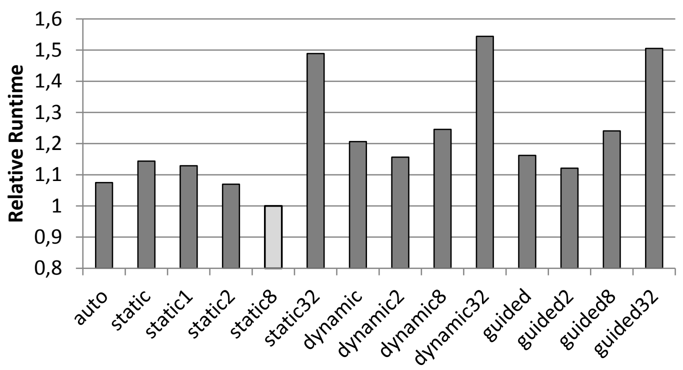
(b) Triangular Matrix Multiplication

Figure 6.7: Initial Experiments, Impact of Program Characteristics

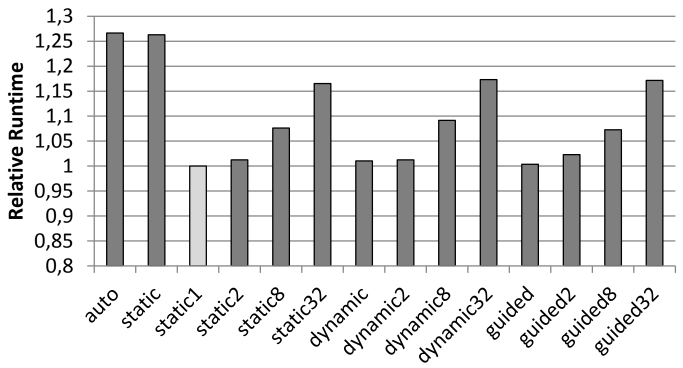
worsened load balance for a static chunk size of 8, leading to this configuration showing the best result. With large problem sizes, the relative overhead of runtime scheduling is much smaller, tough still measurable. The round-robin static scheduling policy “static,1” features acceptable load balance with relatively low overhead, making it the best performing configuration.

Finally, we look at a scenario that has often been neglected in loop scheduling research: the impact of external system load on the execution of a program. While this is an unusual situation in traditional high performance computing (HPC), where a cluster of servers is reserved for exclusive use by one program, it is the default on desktops, workstations and some large shared memory servers. With on-chip parallelism steadily increasing – even on embedded systems – and OpenMP being employed in end-user applications and games [55], we believe that an automatic loop scheduler needs to take this scenario into account.

Figure 6.9 shows the same program configurations as Figure 6.7(b) in two distinct load scenarios (for information on how the load simulation is performed, see Section 6.3.3). With increasing system load more fine-grained



(a) Small problem size (N=160)



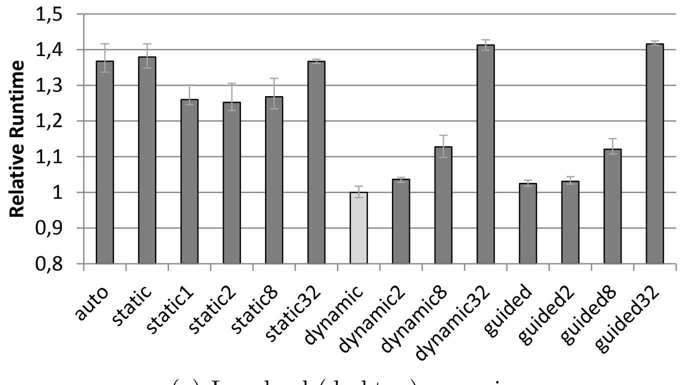
(b) Large problem size (N=1600)

Figure 6.8: Initial Experiments, Impact of Problem Size

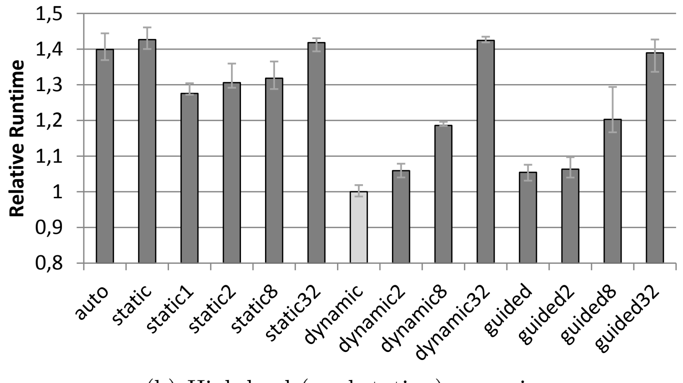
runtime scheduling policies gain a significant advantage of up to 46% compared to the default policy. These figures contain error bars since there was a slightly larger variance in the measurements – particularly for static scheduling – as a result of operating system scheduling behavior.

To summarize: these initial findings guided the design of our loop scheduling in the following ways:

- As per the first set of figures, the automatic loop scheduler clearly needs to be aware of the *program structure*. This is accomplished via compiler analysis.
- However, as the second set of examples shows, just having static information is insufficient. The *problem size* is usually only known at runtime, necessitating integration of static compiler analysis with a runtime system.
- Finally, when exclusive use cannot be assumed, being aware of *external system load* is of utmost importance when selecting a scheduling policy. Thus, loop scheduling needs to consider the system state.



(a) Low load (desktop) scenario



(b) High load (workstation) scenario

Figure 6.9: Initial Experiments, Impact of External Load

6.3.2 Method

Our automated loop scheduling system is based on the Insieme infrastructure. Figure 6.10 outlines its basic structure. After an OpenMP input program is processed by the frontend (1) and converted into the Insieme IR a customized analysis is utilized to to obtain a symbolic *effort estimation function* for each parallel loop body. This parametrized function provides a work-load estimation for a given range of loop iterations. The analysis results are annotated to the program (3) and converted by the backend into target code including those functions in the form of meta-information associated to the individual parallel loops (4). During the execution, whenever a parallel loop is reached, the *loop scheduler* decides on how to distribute the corresponding range of loop iterations among the available resources. This decision is based on the actual iterator range, meta-information forwarded from the compiler (5) and data continuously collected by the runtime's monitoring system (6).

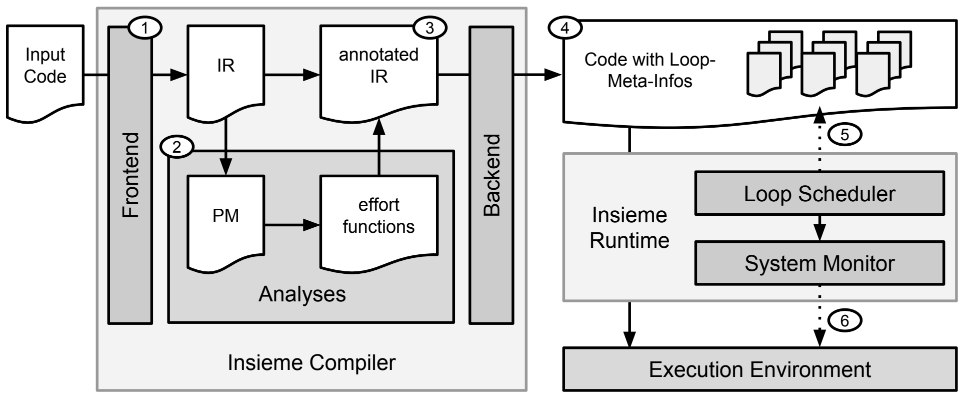


Figure 6.10: An overview of the automated loop scheduling system.

Compiler Analysis

The goal of the compiler analysis is to obtain, for each parallel loop, an *effort estimation function*

$$f_{\text{effort}} \in \mathbb{N}^2 \to \mathbb{N}$$

characterizing the loop body. Given lower and upper iteration bounds a and b, the evaluation of $f_{\text{effort}}(a, b)$ should provide an estimate for the computational cost of the corresponding sub-range of the covered loop.

This effort estimation function is derived in several steps, starting from the parallel loop body B:

1. Enclose B in a *for* loop iterating over the symbolic range $[a, b)$
2. Extract a polyhedral representation of this parametrized loop
3. Initialize the effort estimation function by $f_{\text{effort}}(a, b) := 0$
4. For each statement $s \in \mathbb{S}$ in the polyhedral representation of B:
   (a) Use the barvinok [104] library to obtain a piecewise affine function for the statement's cardinality $f_{\text{card}}(a, b)$
   (b) Weight this function with the effort estimation $\text{eff}(s)$ for the statement, computing $f_{\text{s}}(a, b) := f_{\text{card}}(a, b) * \text{eff}(s)$
   (c) Add the resulting symbolic statement effort to the total effort function by computing $f_{\text{effort}}(a, b) := f_{\text{effort}}(a, b) + f_{\text{s}}(a, b)$
5. Algebraically simplify $f_{\text{effort}}(a, b)$

In step 2, the IR version of the processed loop is converted into a polyhedral representation utilizing the infrastructure introduced in Section 4.5.1. To estimate the overall computation effort of the loop body, step 4 sums up the individual costs of the involved statements. Thereby, for each statement, a symbolic expression describing the cardinality of its iterator domain is obtained (4a) and multiplied by the computation cost of each statement instance (4b). The latter is obtained using the code feature extraction utilities covered by Section 4.3.2 to estimate the number of memory access and floating point operations required to process the given statement. Finally, after summing up the results of the individual statements, a symbolic expression estimating the computational cost of processing the sub-range $[a, b)$ is obtained and converted into the desired function.

In case the loop body can not be converted into a polyhedral representation since it does not satisfy the constraints of a SCoP, an estimate of the workload of a single loop iteration is obtained by applying the feature extraction utility used in step 4b on the entire loop body instead of individual statements. The framework of Definition 4.2 is thereby handling the weighting of nested structures. Experimental data on how commonly this fallback needs to be employed in real programs can be found in the full paper [97].

Compiler Backend

The backend creates for each parallel loop a *work item* (see Section 2.4.1). To pass loop-related meta-information from the compiler to the runtime, an (optional) function pointer of type

```
uint64 effort_estimator(int64 lower, int64 upper)
```

and a scalar fallback value

```
uint64 iteration_effort
```

are additionally attached to each of those work item. For each loop where our analysis was successful, the function pointer is set to a compiler generated C implementation of the deduced effort estimation function, otherwise it is set to `NULL` and the field `iteration_effort` is filled by the value obtained from the fallback analysis.

Runtime Monitoring

The resource monitoring component of the runtime needs to measure the current *external load*, that is, CPU load generated by processes other than the managed parallel program. This is obtained by using the Linux `proc` filesystem. Specifically, the current processes' CPU usage values from `/proc/self/stat` are compared with the system-wide values obtained from `/proc/stat`, and a value between 0.0 and 1.0 representing the total external load across all cores is computed. To minimize the overhead of this method and to increase measurement reliability, this value is cached and updated at most ten times per second. Increasing the update frequency did not improve scheduling performance in our experiments.

Algorithm 6.1 Automatic loop scheduling algorithm.

`lower`, `upper`	lower and upper bound of iteration range
`members`	number of threads in the local thread group
`estimator`	effort estimation function for current loop
`iter_effort`	scalar per-iteration effort estimate for current loop
`load`	current external system load
`MINEFF`	minimum effort for consideration (constant per-system)
`MINLOAD`	minimum load for consideration (constant per-system)

```
1: if estimator available then
2:     estimate = estimator(lower, upper)
3: else
4:     estimate = (upper − lower) ∗ iter_effort
5: end if
6: if estimate < MINEFF then
7:     return immediate
8: end if
9: if load > MINLOAD then
10:     chunk = max((MINEFF/iter_effort) ∗ (1 − load), 1)
11:     return dynamic(chunk)
12: end if
13: if estimator available then
14:     shares = compute_shares(lower, upper, members, estimator)
15:     return balanced(shares)
16: else
17:     chunk = max(MINEFF/iter_effort, 1)
18:     return dynamic(chunk)
19: end if
```

Loop Scheduling Algorithm

All information gathered by the components outlined above is used by the runtime loop scheduler to make a scheduling decision for each individual execution of every parallel loop. The decision algorithm is outlined in Algorithm 6.1 and consists of five major steps:

1. Estimate the workload effort (lines 1-5)
2. Immediately schedule tiny loops if the estimated effort is small (6-8)
3. Check the external load and use an adaptive dynamic schedule if it is greater than a threshold value (9-12)
4. If an effort estimation function is available, use calculated balanced distribution (13-15)
5. Otherwise, assume irregular load and schedule dynamically (16-19)

The result of the algorithm determines the loop scheduling behavior for the current loop execution instance. Three modes are available:

immediate no parameters. Immediately executes the whole loop on the first thread to encounter it.

dynamic one parameter, the chunk size. Works like the standard OpenMP policy of the same name, dynamically distributing chunks of the loop range to requesting threads.

balanced requires an array of floating point values determining the relative starting points of the shares for each member of the work group. For example, [0.0, 0.25, 0.5, 0.75] would implement an equal distribution amongst four threads, while [0.0, 0.6, 0.9, 0.96] assigns progressively smaller chunks to subsequent threads.

The algorithm makes use of the

$$\text{compute\_shares}(\texttt{lower}, \texttt{upper}, \texttt{members}, \texttt{estimator})$$

function. It generates a distribution that tries to assign approximately the same amount of work to each member of the current work group. It first estimates the total effort for the given range [`lower`, `upper`], divides it by the number of work group members, and then uses a binary search to find a suitable chunk for each thread using the estimation function. Though this is usually a very quick process since the estimation function only takes a few cycles to run, the result is cached and reused if the same loop is executed for the same range again.

The parameters `MINEFF` and `MINLOAD` need to be set once per system. We have not yet developed a rigorous method for deducing these automatically. Nevertheless, experience indicates that systems are relatively insensitive regarding the precise values of these parameters, making them easy to tune manually.

6.3.3 Results

All experiments were performed on a SuperMicro 7046GT-TRF server with two Intel Xeon 5650 processors, containing 6 cores (12 hardware threads) each. The system runs CentOS version 5 (kernel 2.6.18) 64 bits. To compile the reference version of the example programs and as a secondary compiler for the code produced by Insieme, GCC version 4.5.3 was used with the -O3 flag set to reflect a production environment. When we refer to a "default" scheduling policy, we specifically mean the default implementation of the version of GOMP [69] included with this version of GCC.

To ensure statistical significance, each experiment was repeated five times, and the median result is reported. In cases where significant statistical variance occurred vertical error bars are used to show the standard deviation. We depict three values per configuration (combination of program and system load state): the default OpenMP behavior, the best result

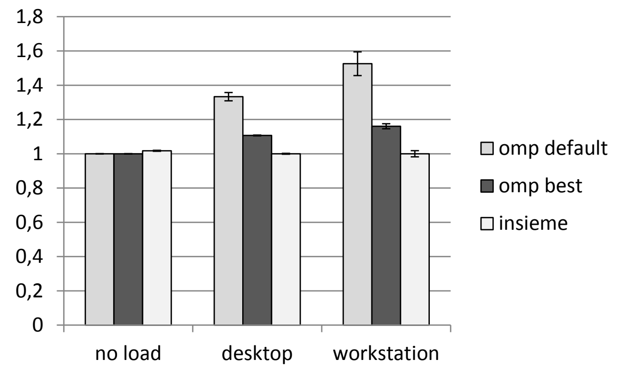


Figure 6.11: Dense matrix multiplication results.

obtained using OpenMP policies for each configuration, and the result obtained by our method. The "best" OpenMP policy is found by exhaustive search across the following settings: [(no change), auto, static, dynamic, guided]. The latter three are tested with the chunk sizes 1, 2, 8 and 32. All values are normalized to the execution time of the best performing version.

External load

Profiles were recorded by monitoring each individual core of a reference system. During experiments, these profiles were replayed by a custom load generator. We used two separate load profiles, a "desktop" profile and a "workstation" profile. The former features generally lower load and short peaks of activity, while the latter shows a higher average load level and fully saturates some cores.

Kernel Experiments

For illustrative purposes, we will apply our method to three small kernels: a dense matrix multiplication, a triangular matrix multiplication, and a pendulum simulation. These represent three major classes of problems. Both the dense and triangular matrix multiplication satisfy the SCoP constraints and can therefore be rigorously analyzed. The former has a flat work profile and is thus ideally suited to static OpenMP scheduling, while the latter has a slanted work profile. Finally, the per-iteration work in the pendulum kernel strongly depends on the input data, hence it can not be covered by SCoP analysis.

Figure 6.11 shows the results for dense matrix multiplication. In the absence of external load, fully static scheduling is ideal for this kernel, and our implementation is 1.7% slower than the best (and default) OpenMP policy. With external load, the default policy is ineffective, and our result improves on the best OpenMP policy by 10% to 15%. The best policy found

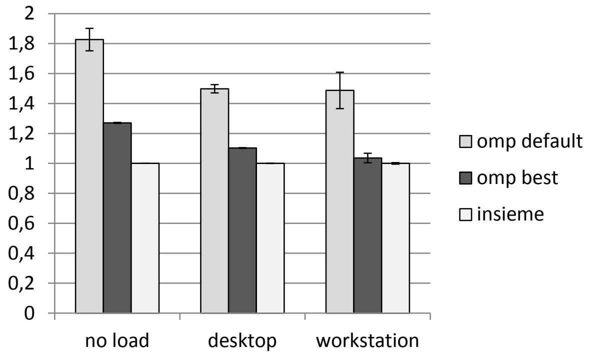


Figure 6.12: Triangular matrix multiplication results.

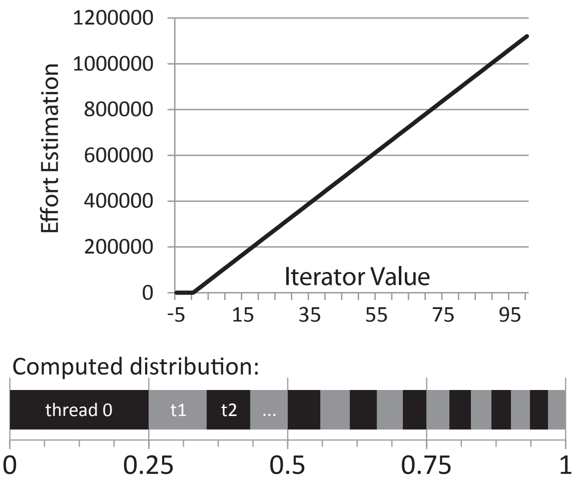


Figure 6.13: Triangular matrix multiplication effort estimation function.

for desktop load is “dynamic,8” while the best policy for the workstation load profile is “dynamic”. The reason for the good result demonstrated by our method is that due to the detection of external load the chunk size is adapted dynamically.

Triangular Matrix Multiplication Kernel Next, we look at the triangular matrix multiplication kernel, which has a more interesting load profile. As Figure 6.12 illustrates, the compiler-assisted workload distribution performed by our method in the unloaded case is very effective, improving performance by 82% compared to the default behavior, and by 27% compared to the best OpenMP scheduling policy, “static,2”.

This improvement over the block-cyclic scheduling can be explained by the effort estimation function generated by our analysis. The per-thread shares computed for 16 threads are shown in Figure 6.13. In the upper part, the effort estimation for each iterator value is plotted: iterations below

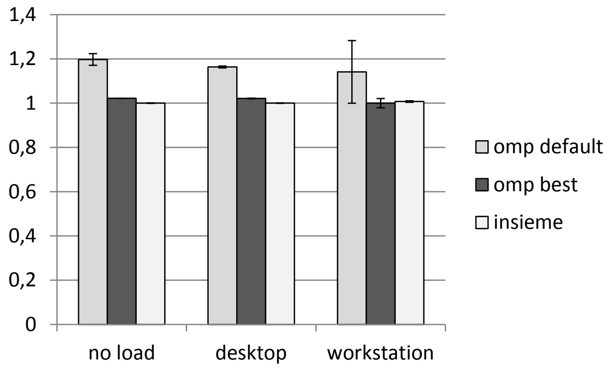


Figure 6.14: Pendulum simulation results.

zero perform no work, above that the amount of effort increases with the iterator value as the lower left triangular matrix rows become progressively wider. For this test case, the best scheduling policy with a loaded system is “dynamic” for both load profiles. Our scheduling is the fastest for both situations, though in the “workstation” case the difference is negligible (3%).

Pendulum Simulation The performance results for the pendulum kernel are depicted in Figure 6.14. This benchmark computes the resting points of pendulae under the effect of magnetic fields, from many starting locations. It is communication-free but has an unpredictable, input data dependent, load imbalance, causing default scheduling to be sub-optimal. For the case with no load, the “dynamic,2” policy is best, while for the other two cases “dynamic” performs best. When the workstation external load profile is active, our method performs slightly (0.7%) worse than the “dynamic” OpenMP policy. For this load profile and the loop effort estimated for this kernel, our scheduler always decides to dynamically distribute a single loop iteration, thus performing exactly the same operation as the “dynamic” policy. The 0.7% difference can be explained by the overhead introduced by our scheduling process.

Real-world Applicability To evaluate our system on real-world applications beyond simple kernels its effect on the benchmarks contained in the NAS Parallel Benchmarks (NPB) [9] suite have been investigated. The results are summarized in Table 6.7.

The “Default” and “Best” columns list the relative difference in execution time achieved by our scheduling system compared to default scheduling (as specified by the benchmarks) and the best scheduling policy found in the search space described earlier. For example, 4.2% in the ft.B/none/default cell means that executing the ft benchmark with no external load and the default scheduling policy took 104.2% of the time the same configuration

		Gain Over		
Name	External Load	Default	Best	Best Config
ft.B	none	4.2%	-0.2%	static,1
ft.B	desktop	21.8%	4.4%	dynamic,2
ft.B	workstation	59.9%	11.2%	dynamic
ep.B	none	14.0%	-1.9%	dynamic,8
ep.B	desktop	3.2%	-0.9%	dynamic
ep.B	workstation	19.7%	3.0%	dynamic,32
bt.B	none	-2.4%	-2.4%	static
bt.B	desktop	70.8%	65.2%	dynamic
bt.B	workstation	-	-	-
cg.B	none	8.4%	3.9%	guided,32
cg.B	desktop	113.4%	111.2%	guided,32
cg.B	workstation	471.3%	451.7%	guided,8
mg.B	none	51.7%	5.3%	dynamic
mg.B	desktop	56.1%	33.0%	dynamic
mg.B	workstation	157.4%	110.8%	dynamic,2
GM	none	13.7%	0.9%	
GM	desktop	48.2%	36.8%	
GM	workstation	94.9%	67.7%	

Table 6.7: Nas Parallel Benchmark performance results.

took using our scheduling system. The bt benchmark with workstation external load could not be completed due to time constraints – the execution time increased disproportionately with increased load across all scheduling policies. The GM values are the geometric means, for each configuration, across all benchmarks.

Note that even with no external load, our method tends to achieve an improvement over default scheduling due to the availability of compiler-deduced meta-information. The average speedup obtained in this setting is 13%. A detailed evaluation of the results can be found in the full publication presenting this loop scheduling approach [97].

6.3.4 Conclusion

Based on the Insieme infrastructure we established an automated OpenMP loop scheduling system that combines advanced compiler analysis with a load-aware runtime system. Polyhedral analyses are used to calculate a parametrized *effort estimation function* for each parallel loop, based on the cardinality of all statements it contains. Executable code for this function is generated by the compiler backend, and invoked at runtime to calculate an ideal balanced schedule or estimate efficient chunk sizes for dynamic scheduling. Additionally, external CPU load is taken into account during the scheduling process.

We evaluated our system on small kernels as well as programs from the NAS Parallel Benchmarks suite, and achieved improvements of up to 82% in the unloaded state, and 471% with heavy external load, compared to default OpenMP scheduling. To estimate the absolute effectiveness of our approach, we performed an exhaustive search over a broad range of standard OpenMP scheduling policies and compared with the best results. Our scheduling frequently improves upon even this tuned result, particularly in scenarios featuring external load. The worst-case performance achieved by our system is within 3% of the best standard OpenMP policy.

6.4 Improved Task Scheduling

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [7]. While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. A central feature of every task-based parallel program that significantly affects both efficiency and scalability is *task granularity* [32]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks.

Very fine-grained, short-running tasks lead to a loss in efficiency compared to sequential execution due to the runtime overhead associated with generating and launching a task, as well as synchronizing its completion with other tasks in the system. On the other hand, coarse-grained, long-running tasks minimize overhead, but are hard to schedule effectively and may therefore fail to scale well on large parallel systems. Previous work in this area has focused mostly on runtime systems or user-controlled cutoffs to manage granularity. Conversely, we devised an approach based on the Insieme infrastructure that combines a multiversioning compiler with a runtime system which adaptively selects from the generated versions. Our goal is to maximize efficiency by increasing task granularity – and thus decreasing overheads – without negatively affecting load balance or scalability.

The content of this section is a summary of previously published work by Thoman, Jordan and Fahringer [99, 100] and derived work [98]. The contributions of the author of this thesis are the implementation of the necessary building blocks for the applied code transformations and the corresponding backend extensions in the Insieme compiler as well as contributions in the design of the overall approach and the decision making component in the runtime system. All of those are covered in detail in Section 6.4.2.

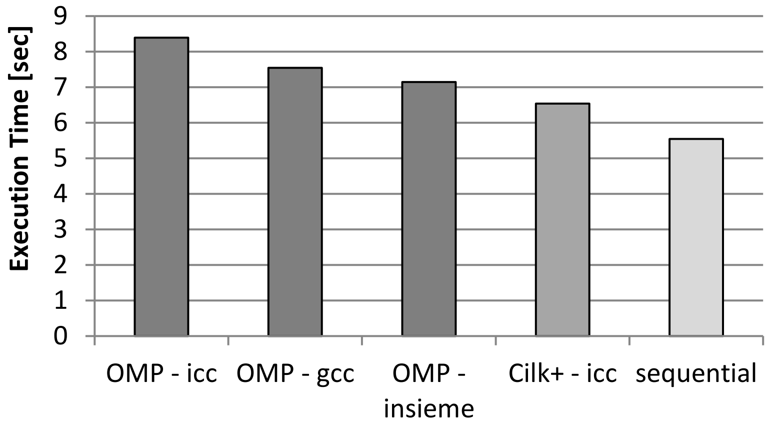


Figure 6.15: Initial Experiments, N-Queens $N = 13$, single-threaded.

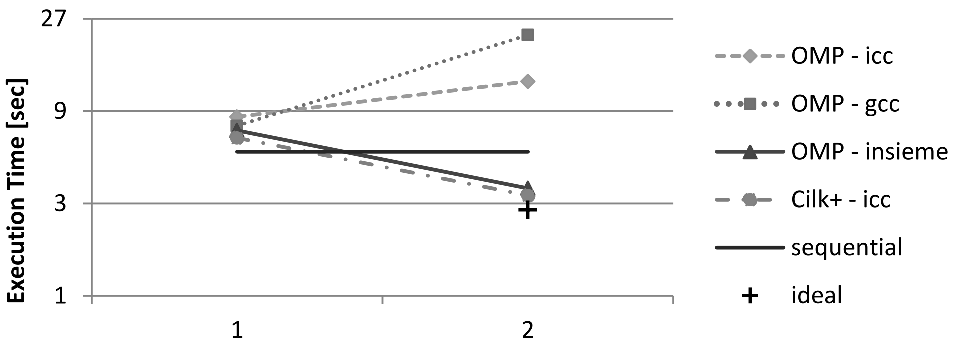


Figure 6.16: Initial Experiments, N-Queens $N = 13$, scaling from 1 to 2 threads.

6.4.1 Motivation

Figure 6.15 shows single-threaded execution times measured for the Barcelona OpenMP Tasks Suite (BOTS) [31] N-Queens benchmark with $N = 13$. For details on the hardware, compiler versions and programs used refer to Section 6.4.3.

The lowest execution time amongst the OpenMP versions is achieved by our compiler and runtime system (Insieme), however, this time is still 28% higher than purely sequential execution. Even the Cilk version, while more efficient than any OpenMP implementation, is 19% slower than the sequential version. Our multiversioning method is designed to address this inefficiency. Throughout this section, when we refer to *inefficient* execution, we mean execution which takes longer than executing purely sequential code (assuming perfect scaling).

Note that the OpenMP runtime systems of ICC [45] and GCC [93] perform special case handling when only a single worker thread is used. This is visible in Figure 6.16, which shows their performance degrading when switching from one to two threads. Further experiments in Section 6.4.3 confirm this behavior, with scaling starting after some initial performance degradation when activating multi-threaded execution. The OpenMP version compiled with Insieme and the Cilk version do not suffer from this issue, however they still induce a relative overhead of about 20% compared to ideal linear scaling from the sequential version. We identified the following potential causes for this inefficiency:

1. Task generation overhead. This includes generating a task structure, populating it with values and enqueuing it.

2. Synchronization primitive overhead (e.g. the OpenMP `taskwait` directive). At the very least, this involves keeping track of all the subtasks launched by each task, and signaling when they are complete.

3. Task library calls. The runtime methods required for tasking are generally implemented in a separate library, and the overhead for their invocation is incurred even if they perform no actual work.

4. Non-inlineable, indirect program function calls. Since the program function implementing a given task needs to be called by the tasking library, a pointer to it is usually passed to the library function. Even if the runtime library decides to directly execute the call, this prevents the benefits – improved instruction scheduling and a reduction in overhead – associated with inlining.

Issues 1 and 2 can be mitigated by a pure runtime approach, e.g. the runtime library can dynamically decide whether to generate a full task structure or directly call the task function. This method is usually referred to as lazy task creation [62]. However, the basic overhead of library function calls (issue 3) and the fact that indirectly called functions in the original program can not be inlined (issue 4) can not be changed at runtime and need to be handled at compile time. This limitation of pure runtime systems motivates our compiler-aided multiversioning approach.

All four potential causes for inefficient execution identified above are directly related to and influenced by the granularity of tasks. The more often individual tasks are generated and synchronized, the higher the impact of the associated overheads on execution time. However, simply increasing the granularity of all tasks is not a solution: such an approach will lead to load imbalance, increasing the probability of workers idling. Therefore, our goal is the generation of different implementations for each task.

6.4.2 Method

Figure 6.17 provides an overview of our approach. The process starts by converting an OpenMP or Cilk program with parallel tasks into our IR

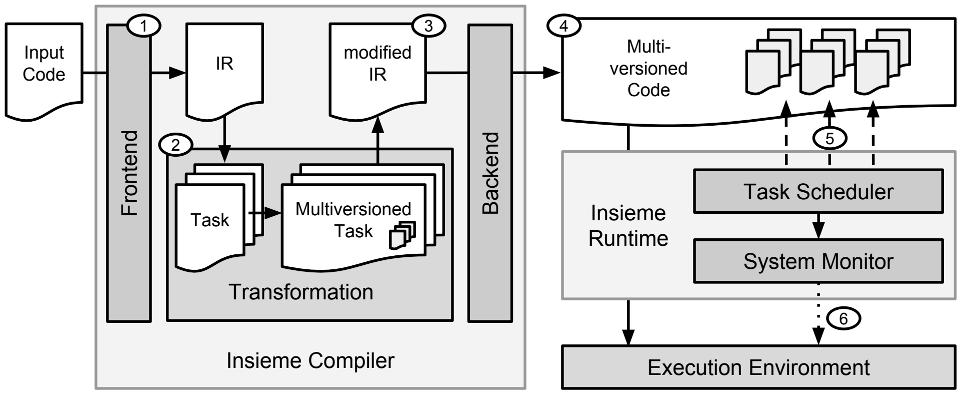


Figure 6.17: Overview of the task scheduling system.

(1). In the next step, each code fragment spawning a task is transformed into a semantically equivalent code fragment composed by multiple different versions of the task (2). The resulting versions replace the original task implementations in the input code (3) and the backend produces target code where each parallel task is represented a *work item* – see Section 2.4.1 – exhibiting multiple implementations (4). During the program execution, whenever a specific task is invoked, a *task scheduling* component selects dynamically among the available implementations (5) based on the current system state (6). In the following the involved components are elaborated in more detail.

Compile-time Multiversioning

During compilation our goal is to generate multiple versions of each parallel task, with varying granularity. This involves a three step process, which may be applied multiple times to further increase the task size. The individual steps are as follows:

1. **Task unrolling**. Replaces each task invocation site with a direct call to the task function, which is subsequently inlined. This can be thought of as a context and parallelism-aware recursive function inlining step. The name *task unrolling* is adapted from Rugina's usage of *recursion unrolling* [90]. Essentially this step is an application of the recursive function unfolding operation introduced in Section 5.2.2.

2. **Sequentialization**. This step focuses on identifying which synchronization primitives – if any – were rendered superfluous by the partial elimination of parallel task invocations due to task unrolling, and removing them. This particular step has been covered in Section 5.3.4 as an example of pattern based transformations.

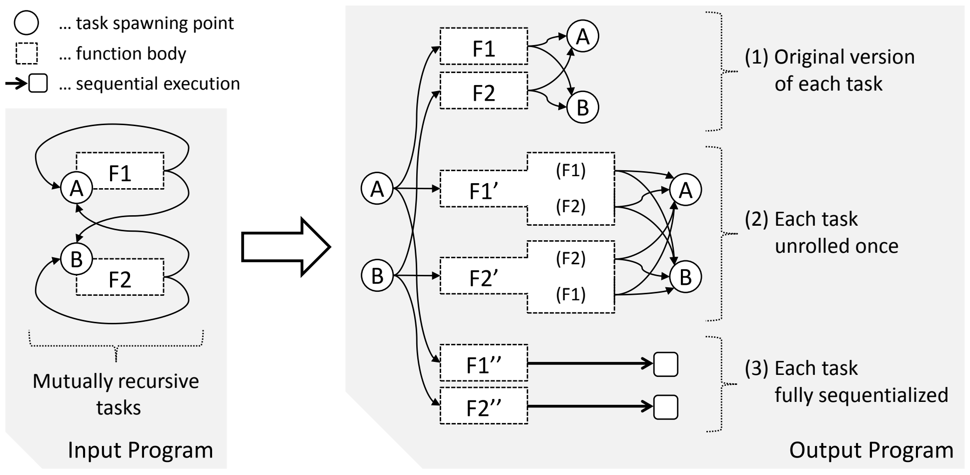


Figure 6.18: Version Generation and Control Flow

3. **Simplification**. The unrolling and sequentialization may have generated code that can be simplified by basic operations such as arithmetic simplification, constant propagation or dead code elimination. Thus, these are performed before any further processing.

The number of generated versions depends on the granularity of the initial tasks and the largest granularity desired. The versions are generated and encoded into the target program in the following order.

1. **Original**. The original version from the input program.

2. N **times unrolled versions**. Starting from $N = 1$. In these versions, only partial sequentialization is performed. Outer task spawning points are removed, but the innermost spawning location is kept. This process is illustrated in detail in a code example described below.

3. **Fully sequentialized version**. In this version all task spawning points are removed and replaced with plain function calls.

Figure 6.18 illustrates the result of generating 3 versions for a mutually recursive task set consisting of two functions $F1$ and $F2$. The original program thus has two task spawning locations, A (which spawns $F1$) and B (spawning $F2$). To improve the clarity of the illustration, these task spawning points have been replicated in the figure, however they are still all referring to the same task.

Version (1) is identical to the original program, except that at each spawning point there is now a choice between 3 distinct implementations of each function. In version (2), consisting of $F1'$ and $F2'$, each recursive task invocation was unrolled once, forming tasks of increased granularity. Clearly, if this version is used, more work is performed between individual task invocations and interactions with the runtime library. Finally, version (3), comprising $F1''$ and $F2''$, is fully sequentialized. Once this version is invoked, no further parallel tasks will be spawned on this branch of the recursive descent.

Code Example

To illustrate the effect of the transformations, consider the following pseudo code fragment implementing a parallel version of the recursive computation of Fibonacci numbers.

```
let fib = (n) → int {
  if(n<2) return n;
  a = spawn(fib(n−1));
  b = spawn(fib(n−2));
  sync;
  return a + b;
}
```

A pseudo-code formulation is used for reasons of clarity and size. It closely re-assembles the familiar IR syntax, yet drops explicit variable declarations and utilizes two additional constructs: `spawn` implies the generation of a new parallel task (corresponding to Cilk's spawn or OpenMP's `#pragma omp task untied`), while `sync` waits for the completion of all launched subtasks (equivalent Cilk's sync or to OpenMP's `#pragma omp taskwait`). Both constructs can be directly replaced by IR constructs – see Section 3.9.

In the first step, first-level task invocations are replaced with in-place calls of the associated functions, producing the following version.

```
let fib = (n) → int {
  if(n<2) return n;
  a = (n2) → int {
    if(n2<2) return n2;
    a2 = spawn(fib(n2−1));
    b2 = spawn(fib(n2−2));
    sync;
    return a2 + b2;
  }(n−1);
  b = (n3) → int {
    if(n3<2) return n3;
    a3 = spawn(fib(n3−1));
    b3 = spawn(fib(n3−2));
    sync;
    return a3 + b3;
  }(n−2);
  sync;
  return a + b;
}
```

Context-sensitive inlining of these calls results in the following, single recursive function.

```
let fib = (n) → int {
  if(n<2) return n;
  if(n−1<2) a = n−1;
  else {
    a2 = spawn(fib(n−1−1));
    b2 = spawn(fib(n−1−2));
    sync;
    a = a2 + b2;
  };
  if(n−2<2) b = n−2;
  else {
    a3 = spawn(fib(n−2−1));
    b3 = spawn(fib(n−2−2));
    sync;
    b = a3 + b3;
  };
  sync;
  return a + b;
}
```

Finally, redundant applications of the `sync` operation are removed using the transformation presented in Section 5.3.4 and arithmetic simplification is applied. This results in the final version given by

```
let fib = (n) → int {
  if(n<2) return n;
  if(n<3) a = n−1;
  else {
    a2 = spawn(fib(n−2));
    b2 = spawn(fib(n−3));
    sync;
    a = a2 + b2;
  };
  if(n<4) b = n−2;
  else {
    a3 = spawn(fib(n−3));
    b3 = spawn(fib(n−4));
    sync;
    b = a3 + b3;
  };
  return a + b;
}
```

Note that the last `sync` call has been eliminated. This process can be repeated N times to generate increasingly larger task sizes.

After all the versions are generated, each version needs to be modified to enable runtime selection. The original version is converted into

```
let fib = (n) → int {
  if(n<2) return n;
  a = spawn(
    pick(fib, fib_u1, fib_seq)(n−1)
  );
  b = spawn(pick(
    pick(fib, fib_u1, fib_seq)(n−2)
  );
  sync;
  return a + b;
}
```

where the `pick` keyword implies a possible choice between semantically equivalent versions, which is deferred to the runtime system. The unrolled version produced above is converted to

```
let fib_u1 = (n) → int {
  if(n<2) return n;
  if(n<3) a = n−1;
  else {
    a2 = spawn(
      pick(fib, fib_u1, fib_seq)(n−2)
    );
    b2 = spawn(
      pick(fib, fib_u1, fib_seq)(n−3)
    );
    sync;
    a = a2 + b2;
  };
  [.. same for b ..]
  return a + b;
}
```

and an unrolled, fully sequential version to

```
let fib_seq = (n) → int {
  if(n<2) return n;
  if(n<3) a = n−1;
  else {
    a2 = fib_seq(n−2);
    b2 = fib_seq(n−3);
    a = a2 + b2;
  };
  [.. same for b ..]
  return a + b;
}
```

This way, the processing of an initial call to the function `fib` is processed in parallel, where the runtime may decide at every step how large the tasks to be processed in the next nested steps should be – depending on the current load distribution in the system.

A more detailed description of the covered transformation steps can be found in the related publications [98, 99].

Runtime Version Selection

The previous section outlined how multiple versions with different granularity and trade-offs are generated in the compiler. This provides the runtime system with an opportunity of making a version choice every time a task is spawned. Making the wrong choice can result in not gaining the desired increase in efficiency, or, at worst, greatly diminishing parallelism – e.g. in case a fully sequentialized version is chosen too early. We considered the following design goals and observations when developing our version selection method:

- At the start of the program, the original (most fine-grained) version of the tasks should be used, since the parallelism available in the system is not yet fully leveraged and load-balancing is a priority.
- The impact of conservative behavior – i.e. using more fine-grained tasks – causes more gradual performance degradation than using tasks that are too coarse grained, potentially leading to some worker threads idling.
- The decision procedure needs to be simple and not introduce large overheads on its own, otherwise it could negate any benefits from multiversioning.
- The decision making process should be distributed – no new synchronization points between worker threads should be introduced to facilitate version selection.

Taking these points into account led to the development of a distributed version selection heuristic based on two parameters that are tracked for each individual worker thread. The first is *task demand*, which keeps track of other worker's unfulfilled attempts to steal tasks from the local worker. The second parameter is the *queue length* of each worker, or how many tasks it currently has available to be executed or stolen.

Task demand is tracked in a surprisingly simple, but effective, manner. The demand is stored as an integer which starts at a positive value equal to the maximum task queue length. Whenever a task is generated by a worker thread, it reduces its own task demand by 1. When a worker attempts to steal from another which has no tasks available, that target worker's demand value is reset to the starting value.

Our version selection procedure is listed in Algorithm 6.2. In conjunction with the demand tracking outlined above, it has the following desirable properties:

- Evaluating the selection function only takes a few dozen cycles, assuming that all the required values are cached.

Algorithm 6.2 Task Version Selection Algorithm (from [98]).

`queue_length`	current queue length
`task_demand`	current task demand
`num_versions`	number of versions generated for current task
`MAX_QUEUE`	maximum queue length (fixed)

output:	0	$\Leftrightarrow$	original task
	$N = 1 \ldots$ `num_versions` $- 2$	$\Leftrightarrow$	unrolled N times
	`num_versions` $- 1$	$\Leftrightarrow$	fully sequentialized

```
1: res = num_versions − ⌈(task_demand/MAX_QUEUE) ∗ num_versions⌉
2: if res >= num_versions − 1 then
3:     if queue_length == MAX_QUEUE then
4:         return num_versions − 1
5:     end if
6:     return num_versions − 2
7: end if
8: return res
```

- The way in which task demand is completely reset if any stealing operation fails, but is only reduced gradually during normal execution, mirrors the earlier observation about the negative performance impact of wrong granularity selection. It makes the expensive case of idle workers unlikely by reacting very strongly to failed stealing attempts.
- Selecting the fully sequentialized version is a step that should only be taken after careful consideration, since it will prevent any further parallelism from being generated on this branch of the recursive descent. Therefore, the heuristic only takes this step if there has been no demand for additional tasks over a large number of spawn points *and* the queue is full.

The choice of the `MAX_QUEUE` parameter has an impact on the effectiveness of this approach. Experimental evaluation has shown that generally, a longer queue is beneficial on systems with a larger number of cores. For the evaluation in Section 6.4.3, `MAX_QUEUE` was set to 32.

6.4.3 Results

In this section we will evaluate the performance impact of our optimization on multiple benchmark programs. We start by describing our measurement methodology and the experimental setup used followed by an in-depth evaluation of a single program, and conclude with an overview of the results of a number of other codes in order to provide a balanced overall impression. More details can be found in the related publications [99, 100, 98].

Experimental Setup

For our experiments we used an Intel-based parallel system, incorporating 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. Table 6.8 summarizes the configuration of this system.

Sockets/	Cache			Software				
Cores	L1d/i	L2	L3	OS	Kernel	GCC	ICC	Insieme
4/40	32K/32K	256K	30M	CentOS 6.3	2.6.32	4.6.3	12.1	g4614502

Table 6.8: Hardware and software platform for experimental evaluation.

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. All experimental runs were repeated five times, and the median runtime is reported.

While the most important comparison for our evaluation is between our compiler with and without our multiversioning method, we also included the results obtained by other platforms to provide a reference for comparison. Table 6.8 includes the exact version number of the compilers used in these comparisons. ICC was used as the backend compiler for the Insieme source to source infrastructure, and its built-in Cilk Plus support was employed to compile Cilk programs. The optimization flag "`-O3`" was enabled for all calls to GCC and ICC.

A Detailed Evaluation

The first program we will evaluate is the N-Queens benchmark included in BOTS [31]. Each task in N-Queens spawns 0 to N child tasks, and the depth of its task invocation trees varies from 1 to N, while not following any simple pattern. The size of individual tasks is relatively small.

Figure 6.19 illustrates the performance of N-Queens using a variety of compilers and implementations. Four OpenMP versions are shown: GCC, ICC and Insieme with ("taskopt") and without ("insieme") task optimization. Additionally, we included the results of a Cilk version and a fully sequential version without any parallel language primitives. The execution time is presented in a log-log plot to improve readability. An efficiency plot is provided in Figure 6.20, which compares the execution times of the parallel versions against ideal scaling from the sequential version.

In terms of OpenMP results, it is clear that the task granularity in this benchmark is too small to be handled effectively by GCC's GOMP implementation. ICC shows the same behavior that was already partially observed in Section 6.4.1 – execution time increases when going from a single-threaded to a multi-threaded setup. However, starting from two threads performance

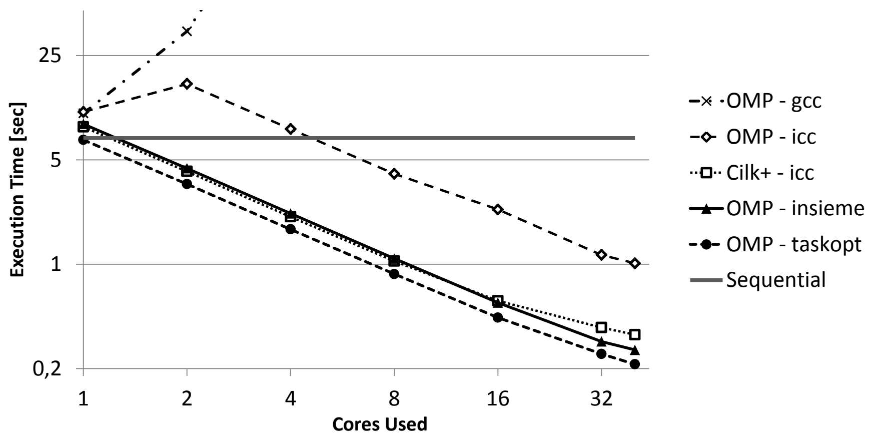


Figure 6.19: N-Queens benchmark results, $N = 13$, execution time.

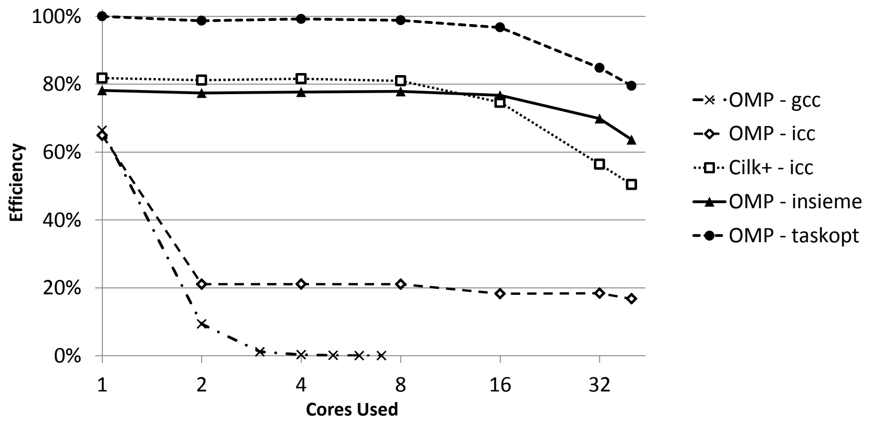


Figure 6.20: N-Queens benchmark results, $N = 13$, efficiency.

scales relatively well up to 40. Since both of these OpenMP implementations seem ill-equipped to handle very fine-grained tasking well, we also included a Cilk version, which has previously been shown to provide better scaling for fine-grained tasks [73]. Indeed, this implementation performs better in the single-threaded case and scales more smoothly to multiple cores than the GCC and ICC OpenMP versions.

Using Insieme to compile the OpenMP input program results in performance that is comparable to Cilk for up to 16 cores, and scales slightly better beyond this amount. However, a comparison with the fully sequential version indicates that even the Insieme OpenMP version and the Cilk version lose around 20% of performance to overheads incurred due to parallelization. When our task optimization is activated, this overhead is effectively avoided. Even more importantly, this significant reduction in overhead is achieved without negatively affecting the scalability of the program. Performance

	Improvement over Best Alternative using N Cores					
Code	N=1	2	5	10	20	40
Queens	27.92%	27.52%	17.78%	26.64%	25.35%	24.91%
Fib	26.43×	32.17×	37.90×	58.15×	86.69×	150.36×
Sort	5.61%	5.47%	6.43%	7.47%	7.88%	8.11%
Strassen	3.54%	7.72%	5.77%	10.08%	7.55%	7.52%
Stencil	20.87%	33.49%	39.17%	47.97%	45.50%	28.29%
Floorplan	36.76%	31.25%	22.62%	21.06%	20.52%	27.68%
FFT	1.84%	4.84%	3.48%	3.93%	10.16%	33.21%
QAP	2.11×	2.66×	2.80×	3.59×	4.68×	6.13×

Table 6.9: Benchmark Results

compared to our implementation without task optimization is improved by 22% to 28% across all measured core counts, with a 25% increase at the full 40 cores.

Compared to the fully sequential version, our approach achieves an efficiency above 99% up to 8 cores, 97% at 16 cores, 85% with 32 cores and 80% at 40 cores. Using the full system (40 cores), our implementation with task optimization improves N-Queens performance by 56% compared to the best competing implementation (Cilk).

Further Benchmarks

Table 6.9 summarizes our benchmark results. It includes measurements for the N-Queens benchmark presented above, as well as a number of additional programs from the BOTS benchmark suite. The values represent the relative improvement achieved using adaptive granularity control, compared to the best result among the other three versions (GCC, ICC and unoptimized Insieme version). The data demonstrates the significant performance improvement that can be obtained utilizing the presented technique. A lot more detailed results and descriptions can be obtained from the related publications [99, 100].

6.4.4 Conclusion

We have presented a fully automatic, adaptive approach to parallel task granularity control which goes beyond what can be achieved by improving either just a runtime system or focusing only on compilation. By combining a compiler which performs task multiversioning with a runtime system that adaptively selects from these versions, we were able to minimize parallel runtime overhead even for very fine grained tasks. Our method uses a novel combination of compiler transformations to build an optimized set of semantically equivalent task versions which differ in granularity. The availability of this set of implementations in the compiled program in turn enables our runtime heuristic to adjust the amount of tasks generated, while incurring even less overhead than a traditional lazy task creation system with cut-offs.

Evaluating our proposed method across a set of eight benchmarks has shown that our optimization is widely applicable, and that the magnitude of these improvements is related to the task granularity of the input program. For programs with relatively coarse-grained tasks, execution times are reduced by 5% - 10%, while we can achieve improvements of a factor of 6 or more compared to the best competing implementations in fine-grained test cases. Benchmark results also demonstrate that our runtime selection heuristic successfully ensures that scalability (up to 40 cores) is not negatively affected by adaptive task granularity adjustment. Crucially, our adaptive granularity control scheme improves performance in all tested benchmarks and for any given number of cores.

6.5 Additional Insieme Applications

The work based on the Insieme infrastructure listed so far focuses on parallel programs targeting shared memory platforms by Thoman and Jordan – since those have been the main focus of the author of this thesis. However, the Insieme infrastructure is utilized for a variety of different applications by other project members. A few of those shall be enumerated to conclude this chapter.

Related research is focusing on:

- *OpenCL:* e.g. researching tools automatically converting OpenCL kernels targeting single accelerator devices into kernels to be distributed among multiple devices including facilities for managing the load distribution in heterogeneous environments by Kofler and Grasso [56, 36].
- *MPI:* e.g. static analysis and compiler optimizations for tuning the utilization of MPI communication primitives by improving the utilization of caches by Pellegrini [78]
- *Energy:* e.g. the modeling of energy consumption of codes [42] as well as the integration of energy into the multi-objective auto-tuning framework presented in Section 6.2 by Gschwandtner [41]

A comprehensive, regularly updated list of research conducted based on Insieme project is listed on the project web page [29].

6.6 Summary

In this chapter a variety of applications researched and developed based on the unique capabilities of the Insieme infrastructure have been described.

The covered applications range from a novel multi-objective auto-tuning solution obtaining code variations manifesting optimal trade-offs between various objectives (Section 6.2), over compiler aided loop scheduling techniques capable of achieving significant performance improvements compared to state-of-the-art alternatives (Section 6.3), to a combination of compiler and runtime components automatically adapting the granularity of parallel tasks, resulting in a significantly faster program execution and improved scalability compared to alternative, state-of-the-art solutions (Section 6.4). Additional research activities based on the Insieme infrastructure, including tuning utilities for OpenCL and MPI based applications have been briefly summarized in Section 6.5. Together, all those applications provide substantial evidence that a novel, unified, high-level, parallelism aware source-to-source compiler infrastructure as our own provides a valuable platform for future research and the development of tools in the area of parallel languages and optimizing compilers – as claimed by this thesis's hypothesis.

Chapter 7

Conclusion

Developing applications efficiently utilizing parallel hardware is complex and time consuming. Besides the challenges related to actually identifying parallelism within algorithms, parallelism needs to be explicitly revealed utilizing specialized APIs, providing features that are typically beyond the scopes of the underlying programming languages. Consequently, those constructs are hardly covered by the optimization steps employed by the utilized compiler infrastructures. Thus, the required load management and tuning effort is left to the developer. Furthermore, the utilized APIs and their primitives are commonly restricted to specific types of target architectures, e.g. shared or distributed memory systems or GPUs, leading to a lack of portability of the resulting, parallel programs.

This thesis is based on the hypothesis that a novel, high-level, parallelism aware source-to-source compiler infrastructure can open up a whole new level of influence for compiler and runtime system components that can be utilized for mitigating, or even eliminating, contemporary issues encountered in the context of programming parallel systems. Such an infrastructure, offering the foundation for the development of these kind of utilities, has been presented in this thesis. Its main objective is to off-load workload and system management responsibilities from the software developer to the compilation tool chain. By closely integration static compiler features with dynamic load management and monitoring capabilities, an environment for the necessary tuning steps is established.

The foundation of the system is laid by unified representations for programs – one for the static compilation process, another for the runtime system. The former has been extensively elaborated in this thesis. It provides a unified, parallelism aware, high-level, holistic, language independent intermediate language for the representation and manipulation of coarse-grained parallel programs. Based on its concise structure and its formalized semantic, it provides a valuable platform for research in the area of parallel languages and associated optimizing compilers.

Based on the central intermediate language, frameworks and utility have been established, providing the necessary ingredients for conducting research utilizing the Insieme infrastructure. Among those are essential components like frontends and backends enabling the system to handle real-world codes. Additionally, various tools and frameworks for conduction and developing program analyses and transformations have been established on top of the Insieme IR and elaborated in detail in this thesis. Furthermore, example applications utilizing those capabilities for building tools supporting developers on implementing efficient, portable, scalable and automatically tuned programs for a variety of complex, contemporary architectures have been presented.

7.1 Contributions

A primary contribution of this thesis is the unique design of the *Insieme infrastructure*, including its compiler and runtime components as well as the program models utilized within those. Together they provide an infrastructure for the automated analysis, transformation and tuning of parallel codes as described in Chapter 2.

Although the *runtime program model* forming the foundation of the runtime system has only been briefly covered in Section 2.4, it provides a powerful foundation for dynamic load and resource management components steering the execution of a program. By abstracting the underlying hardware infrastructure utilizing *workers* and *memory blocks* as well as decomposing the original monolithic input program into *work* and *data* items, the runtime is enabled to effectively influence the execution of a program by managing the workload and data distribution. Furthermore, meta-information forwarded by the compiler and multiversioned implementations of the individual work items provide additional opportunities for the runtime system to positively influence the execution of a program.

The major part of the thesis, and the biggest contribution, is INSPIRE – the *Insieme Parallel Intermediate Representation* – a high-level intermediate language offering a unified parallel model comprising a small, fixed set of parallel primitives. It exhibits a number of novel features and properties exceeding the capabilities of conventional compiler IRs:

- **Parallel IR:** unlike conventional IRs, treating parallel APIs as ordinary libraries, the Insieme IR explicitly incorporates parallel language constructs in its design

- **Suitable level of abstraction:** by constituting a high-level, concise, language independent intermediate language, INSPIRE does not suffer from the narrow scope of conventional low-level IRs reassembling assembly like instructions or the restraining complexity of existing, high-level source-to-source compiler IRs semantically bound to their input languages

- **Global perspective:** by eliminating the boundaries of individual translation units, INSPIRE based program representations are not limited to by their scope, enabling whole program analyses and transformations. In particular since parallel control flows are neither confined to individual procedures or translation units, this design decision provides increased insight and control over processed applications and satisfies a necessary prerequisite for handling real-world parallel applications and for the support of a variety of powerful transformations.

- **API and Hardware abstraction:** due to its language independence and capability of modeling parallel constructs offered by hardware-specific APIs, INSPIRE is not restricted to specific types of parallel APIs and target systems. Instead the Insieme IR represents the parallel structure of an application utilizing an abstract, universal, parallel machine, constituted by the formalization of its semantic defined in Section 3.7.

- **Support for hybrid input codes:** input programs utilizing multiple different parallel APIs to express the parallel structures of their components are encoded utilizing a unified set of primitives, resulting in a program where all parallel operations are coordinated by a single entity. Essentially, the system provides means for a compiler supported, unified implementation of arbitrary parallel APIs.

- **Support for composing parallel codes:** by utilizing a single control entity steering the parallel execution of the entire program and supporting nested parallel constructs, the management problem of coordinating individually parallelized software modules is effectively circumvented.

- **A unified research infrastructure:** finally, the Insieme IR and its associated tool set provide a novel platform for researching issues related to the development, optimization and tuning of parallel applications. The spectrum ranges from input languages and parallel APIs over static code generation, analysis and transformation steps to dynamic program steering and tuning utilities.

Additionally, this thesis contributed tools for analyzing and transforming parallel programs encoded utilizing the Insieme IR. In particular the *constraint based analysis* framework covering all IR constructs and a variety of language extensions represents a comprehensive foundation for developing static program analyses targeting parallel programs. Its broad coverage of input codes and dynamic language features based on the concise, functional nature of the underlying IR and its unprecedented support of parallel constructs at the given scale are novel in the field of source-to-source compilers.

Furthermore, the establishment of declarative transformation utilities utilizing the tree-based nature of the Insieme IR and a tool set for composing arbitrary code transformations simplifies the task of developing prototypes of transformations and auto-tuning solutions as they are required for conducting corresponding research.

Finally, based on the Insieme infrastructure, additional contributions regarding the automated, compiler based improvement of the performance and/or scalability of parallel programs have been made that exceed the capabilities of comparable state-of-the-art solutions. Those include the establishment of a generic, *multi-objective auto-tuning framework* and solutions for the automated determination of *loop scheduling policies* and the transparent, continuous adaptation of the *granularity* of recursively nested tasks as covered in Chapter 6.

Together, all those contributions provide substantial evidence for the validity of the thesis's hypothesis that unified, parallelism-aware, high-level programming models utilized by compiler tool chain utilities provide a valuable foundation for the expansion of the capabilities of those tools towards the automated coordination and tuning of parallel programs.

7.2 Future Work

The Insieme infrastructure is continuously maintained and extended by the members of the *Distributed and Parallel Systems* group of the computer science department at the University of Innsbruck. As such, new features and applications are constantly developed.

Among the obvious extensions are additions to the existing tool set. Additional analyses and transformations may be researched and added to the system. For instance, the fact that the Insieme compiler obtains a whole-program perspective on input codes enables transformations which can not be supported by conventional, translation unit based compilers. In particular, transformations targeting the layout of data structures can be implemented – a feature which could definitely contribute to increase data locality or reduce the amount of data to be moved throughout a parallel system, yet can hardly by realized by a conventional compiler infrastructure in case dependencies are crossing translation unit boundaries.

Another line of research to be followed could be the investigation of improved auto-tuning capabilities. The effective composition of code transformations to satisfy optimization objectives remains an open problem which has been further extended by the flexibility of our system's transformation scripting facilities enabling transformations to be targeted on individual code fragments. The potential of this increased flexibility as well as means to deal with the associated enormous search space may be valuable objectives of future research.

Another direction to follow could be program analysis tools enabling developers to identify errors and potential performance issues within their applications utilizing compiler based analyzing techniques. Properly utilized, they may lead to semi-automated program tuning processes combining the insight and creativity of the human developer with the precision and analytical power of compilers.

On a related matter, new prototypes of parallel languages and APIs depending on increased compiler support may be investigated. Cumbersome tasks including the generation of code moving data objects between address spaces, the conversion of program code into code to be processed by accelerators, or load balancing operations may be gradually off-loaded to the compiler which, in turns, provides feedback to the developer regarding encountered obstacles. Furthermore, advanced features including failure recover operations and performance monitoring may be transparently incorporated. The resulting, semi-automated tool chain would combine the strengths of human developers with the mechanical power of compilers and the capabilities of sophisticated generic load management components, monitoring facilities and failure recovery systems developed by experts, thereby exceeding the capabilities and productivity of conventional compiler or library based solutions.

Appendices

Appendix A

The Insieme Sources

Insieme is an open source project maintained by the *Distributed and Parallel Systems* Group of the computer science department at the University of Innsbruck. An up to date version of the sources is hosted on `GitHub` under

`https://github.com/insieme/insieme`

and may be freely accessed, forked, modified and extended. In this appendix a brief overview on the organization of the codes shall be provided.

A.1 The Directory Structure

The root directory of the project contains four directories:

- **code** . . . contains the code of the various modules of the Insieme infrastructure
- **docs** . . . latex based documentation files
- **scripts** . . . various scripts for setting up a build environment for the project; in particular for handling third party library dependencies
- **test** . . . a collection of 400+ (parallel) programs utilized for running integration tests on the Insieme infrastructure and as input for experimental evaluations

Beside those directories, a `cmake` script for establishing a build environment is provided as well as a `readme` manual describing its usage.

A.2 The Modules

In the code directory the various components of the system are organized into several modules. Those include

- **core** ... the Insieme compiler core comprising basic data structures and utilities including
  - the implementation of the IR structure (see Section 3.10)
  - language extensions and their infrastructure (Section 3.8)
  - the visitor infrastructure (Section 4.2)
  - the node mapper infrastructure (Section 5.2.1)
  - type checker utilities (Section 3.5)
  - validity checks (Section 3.6)
  - the basic manipulation tool box (Section 5.2.2)
  - the pattern matcher implementation (Section 5.3)
  - simple, flow-insensitive analyses (Section 4.3)
  - an IR `builder` providing a unified façade for constructing IR structures
  - a `pretty printer` converting IR structures into a human readable format, similar to the syntax utilized throughout this thesis
  - an IR `parser` capable of converting string based code fragments using the syntax utilized throughout this thesis into IR structures
  - a set of IO operations storing and retrieving IR structures to and from binary streams, in particular files
- **frontend** ... the Clang based C/C++ frontend implementation offering a plug-in system for handling parallel APIs and language extensions including OpenMP, Cilk, OpenCL and MPI. Some of those, like the rest of the system, are still under development (Section 2.3 and 3.9).
- **backend** ... an infrastructure for assembling and customizing Insieme compiler backends supporting the generation of runtime dependent or independent C/C++ output code as well as OpenCL kernel codes (Section 2.3).
- **analysis** ... collection of analysis frameworks based on the Insieme IR, including
  - the feature extraction framework (Section 4.3.2)
  - the CBA framework (Section 4.4)
  - the polyhedral model based analysis system (Section 4.5)
  - the dynamic program analysis system (Section 4.6)

  as well as various analyses built on top of those.

- **transform** . . . a collection of transformations and associated utilities including

  - manually encoded transformations (Section 5.2)
  - pattern based transformations (Section 5.3)
  - polyhedral model based transformations (Section 5.4) and
  - the transformation framework primitives (Section 5.5)

  as well as various transformations built on top of those.

- **driver** . . . a set of applications build based on the functionality offered by the other modules (see Chapter 6). Those include, a drop-in replacement for GCC (`insiemecc`) to be utilized within `make` files, a minimal `demo` application demonstrating the utilization of the various modules to build new applications as well as a integration test runner and a few additional Insieme based utilities.

- **runtime** . . . the Insieme runtime system implementation including, among others, the following components:

  - work and data item management (Section 2.4.1)
  - worker and memory management (Section 2.4.2)
  - scheduling operations (Chapter 6)
  - monitoring facilities

  For more details on those components see the PhD thesis covering the Insieme runtime system [98].

- **meta_information** . . . a module for forwarding information between the compiler and the runtime system (Chapter 6)

- **utils** . . . a collection of generic C++ utilities to be utilized by the various modules to simplify the implementation of the features, including container utils, functional utils, meta-programming utilities and generic components including e.g. constraint solvers (Section 4.4.3)

Each module contains three sub-directories:

- **include** . . . the header files providing prototypes and type definitions to be utilized by others to access the offered functionality

- **src** . . . the implementation of the offered functionality

- **test** . . . an extensive list of unit tests evaluating the proper operation of the offered functionality

In addition to providing means for automated unit testing of the implementation, the test cases included in the `test` directory also provide a valuable source of examples demonstrating the utilization of the offered functionality. Furthermore, they promote *test-driven development* – the principle development process utilized by the Insieme project.

List of Symbols

Symbol	Description	Page
$\mathcal{A}_{(s,m,d)}$	polyhedral model access function	292
B	set of basic blocks	186
$\mathbb{B}$	the set $\{\text{true}, \text{false}\}$	-
$\mathbb{C}$	set of class types	163
$\mathcal{C}$	set of channel states	115
$\mathcal{C}$	set of call context strings	203
$\mathcal{C}$	set of constraints	218
D	set of data items	27
D	function assigning domain to type	68
$\mathcal{D}$	function computing type domain constraints	68
$\mathcal{D}_s$	polyhedral model iterator domain	291
d_ϕ	function computing repetition-depth	317
dom	computes the domain of a partial mapping	64
$\mathbb{E}$	set of IR expressions	53
$\mathbb{E}_v$	set of valid IR expressions	86
$\mathcal{E}$	universal environment	89
F	function collecting free type variables	83
F_{var}	function collecting free variables	83
F_{rec}	function collecting free recursive variables	84
F_{brk}	function collecting free break statements	85
F_{ret}	function collecting free return statements	85
FV	function obtaining free variables of term	218
$\mathcal{F}$	set of node filters	335
G	function collecting generic type variables	64
$\mathcal{H}$	abstract set of system resources	30
I	set of low-level program instructions	186
I	set of data index values	235
$\vec{I}$	iteration vector	290
$\mathbb{I}$	set of identifiers	49
$\mathbb{IV}$	set of intermediate variables	91
$\mathbb{IE}$	set of intermediate expressions	91
$\mathbb{IE}_{gsync}$	set of global synchronizing expressions	93

Symbol	**Description**	**Page**
$\mathbb{IE}_i$	set of irreducible intermediate expression	93
$\mathbb{IR}$	set of all IR structures	178
$\mathbb{IS}$	set of intermediate statements	91
$\mathcal{I}$	set of symbol interpretations	89
$\mathcal{I}$	set of node instances	201
$\mathcal{K}$	set of node types	302
L	set of property space values	187
$L_{f(...)}$	forest based property space constructor	241
$L_{t(...)}$	tree based property space constructor	235
$L_{\lvert...\rvert\leq n}$	limited power-set property space constructor	261
$\mathcal{L}$	set of memory locations	135
$\widehat{\mathcal{L}}$	set of abstract memory locations	272
$\mathcal{L}$	set of node labels	206
MGS	function computing the most general substitution	75
MGT	function computing the most general type	73
$\mathbb{M}$	set of pattern variable matches	317
$\mathcal{M}$	set of memory locations	271
$\mathcal{M}$	set of node mapper	303
$\mathbb{N}$	set of natural numbers including 0	-
η	constant referencing no memory location	135
$\widehat{\eta}$	abstract constant referencing no memory location	272
P_a	property space of the arithmetic analysis	263
P_b	property space of the boolean analysis	266
P_r	property space of the reference analysis	272
$\mathbb{P}$	set of valid IR programs	87
$\mathcal{P}$	set of data paths	135
$\mathcal{P}$	set of program points	206
$\widehat{\mathcal{P}}$	set of abstract data paths	272
$Poly$	set of polynomials constructor	262
R_D	set of resource requirement functions	30
$\mathbb{R}$	set of real numbers	-
$\mathcal{R}$	set of ranges	111
$\mathcal{R}$	set of references	135
$\mathcal{R}$	set of thread regions	208
$\widehat{\mathcal{R}}$	set of abstract references	272
$\mathbb{S}$	set of IR statements	58
$\mathbb{S}_v$	set of valid IR statements	87
$\mathcal{S}$	set of program states	94
$\mathcal{S}$	set of synchronization points	207
$\mathbb{T}$	set of IR types	50
$\mathbb{T}$	set of universal trees	312
$\mathbb{T}_c$	set of closed IR types	64

Symbol	**Description**	**Page**
$\mathbb{T}_g$	set of generic IR types	64
$\mathbb{T}_{gvar}$	set of generic type variables	64
$\mathbb{T}_v$	set of valid IR types	64
$\mathbb{T}_\rightarrow$	set of function types	52
$\mathcal{T}$	set of thread addresses	94
$\mathcal{T}$	set of thread contexts	204
$\mathcal{T}$	set of transformations	334
$\mathcal{T}_s$	polyhedral model schedule function	291
$\mathbb{U}$	pattern variable universe	317
$\mathbb{V}$	set of IR variables	52
$\mathbb{V}$	set of pattern variables	312
$\mathbb{V}_{rec}$	set of recursive variables	52
$\mathcal{V}$	set of data item values	28
$\mathcal{V}$	the universal value set	88
$\mathcal{V}$	feature value set	178
W_S	set of work item execution states	30
W_D	set of work item description	30
$\mathcal{W}$	set of abstract worklist instances	218
$\mathbb{Z}$	set of integers	-
Φ	set of tree patterns	312
Ψ	set of list patterns	312
Δ	set of tree generator expressions	322
Σ	set of list generator expressions	322
Υ	set of value generator expressions	322
Γ_Δ	function generating trees	322
Γ_Σ	function generating lists of trees	322
Γ_Υ	function generating values	322
ϵ	empty mapping	64
ϵ	feature extraction function	178
π	data index projection operator	235
$\perp$	empty data path	135
$\perp$	bottom element of a property space	187
$\perp$	constant denoting failed transformation	334
$\top$	top element of a property space	187
$\times$	Cartesian product set constructor	-
$\rightharpoonup$	partial mapping constructor	64
$\mapsto$	constructor for an element of a mapping	64
$\equiv$	definitionally equal types	70
$<:$	subtype relation	71
$\rightarrow$	program state transition	95
$\rightarrow_a$	thread address based state transition	97
$\rightarrow_e$	environment based state transition	97

Symbol	**Description**	**Page**
$\rightarrow_r$	reduction based state transition	97
$\rightarrow_s$	sequential program state transition	95
$\sqcup$	feature aggregation function	178
$\uplus$	union of disjoint sets	-
$\bigsqcup$	one-of-a-set combination operator of a property space	187
$\sqcup$	binary one-of-a-set combination operator of a property space	187
$\sqcup$	a data index union operator	235
$\bigsqcap$	all-of-a-set combination operator of an extended property space	231
$\sqcap$	binary all-of-a-set combination operator of an extended property space	231
$\sqsubseteq$	partial order on property space	187
$\sqsupseteq$	partial order on property space	187

List of Acronyms

Acronym	Description	Page
API	Application Programming Interface	-
AST	Abstract Syntax Tree	-
CBA	Constraint Based Analysis	194
CFG	Control Flow Graph	186
CPU	Central Processing Unit	-
DAG	Directed Acyclic Graph	-
DFA	Data Flow Analysis	186
DSL	Domain-Specific Language	-
GCC	GNU Compiler Collection	7
GPGPU	General Purpose Graphics Processing Unit	-
GPU	Graphics Processing Unit	-
HPC	High Performance Computing	-
INSPIRE	Insieme Parallel Intermediate Representation	21
IR	Intermediate Representation	6
IRSPM	Insieme Runtime System Program Model	27
LLVM	formerly: Low Level Virtual Machine	8
MESI	Modified, Exclusive, Shared or Invalid protocol	27
MPI	Message Passing Interface	-
NUMA	Non-Uniform Memory Access	1
PGAS	Partitioned Global Address Space	-
PM	Polyhedral Model	289
RISC	Reduced Instruction Set Computing	-
SIMD	Single Instruction Multiple Data	-
SMP	Symmetric Multi-Processing	-
SMT	Simultaneous Multi-Threading	-
VLIW	Very Long Instruction Words	-

List of Figures

List of Tables

List of Definitions

List of Examples

Bibliography

[1] C++ support for clang. http://clang-analyzer.llvm.org/dev_cxx.html. (accessed January 6, 2012).

[2] Clang static analyzer. http://clang-analyzer.llvm.org.

[3] clang: a C language family frontend for LLVM. http://clang.llvm.org, October 2012.

[4] The program transformation wiki, 2013. [Online; 30-May-2013].

[5] Pluto - an automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net/, 2014.

[6] G. Aigner, A. Diwan, D.L. Heine, M.S. Lam, D.L. Moore, B.R. Murphy, and C. Sapuntzakis. An overview of the suif2 compiler infrastructure. *Computer Systems Laboratory, Stanford University*, 2000.

[7] Krste et al. Asanovic. The landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, 2006.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[9] D. H. Bailey, E. Barszcz, J. T. Barton, et al. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[10] Rajkishore Barik, Zoran Budimlic, Vincent Cavé, Sanjay Chatterjee, Yi Guo, David M. Peixotto, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The habanero multicore software research project. In Shail Arora and Gary T. Leavens, editors, *OOPSLA Companion*, pages 735–736. ACM, 2009.

[11] M.M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 200–209. ACM, 2010.

[12] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. *Languages and Compilers for Parallel Computing*, pages 209–225, 2004.

[13] Cédric Bastoul. Generating loops for scanning polyhedra: Cloog users guide. *Polyhedron*, 2:10, 2004.

[14] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel Processing Letters*, 15(1-2):3–18, 2005.

[15] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[16] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jrme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben. Mogensen, DavidA. Schmidt, and I.Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer Berlin Heidelberg, 2002.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, 1995.

[18] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[19] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[20] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[21] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[22] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par 2004 Parallel Processing*, pages 292–303. Springer, 2004.

[23] K.D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2001.

[24] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[25] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[26] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *PACT*, pages 375–386. IEEE, 2013.

[27] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[28] Vivek Sarkar Deepak Majeti. Heterogeneous habanero-c (h2c): A portable programming model for heterogeneous processors. .

[29] The Insieme development team. Insieme compiler and runtime infrastructure. `http://insieme-compiler.org`.

[30] Gabriel Dos Reis and Bjarne Stroustrup. A principled, complete, and efficient representation of c++. *Mathematics in Computer Science*, 5(3):335–356, 2011.

[31] Alejandro Duran, Xavier Teruel, et al. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP*, pages 124–131, 2009.

[32] David N. Turner (ed), Hans Wolfgang Loidl, and Kevin Hammond. On the granularity of divide-and-conquer parallelism. In *Glasgow Workshop on Functional Programming*, pages 8–10. Springer-Verlag, 1995.

[33] G. Fursin, Y. Kashnikov, A.W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost gcc: machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[34] Kirsten Lackner Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for cml. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *ICFP*, pages 38–51. ACM, 1997.

[35] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[36] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *PPOPP*, pages 281–282, 2013.

[37] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[38] Khronos Group. Spir 1.0 specification for opencl, 2012.

[39] The Portland Group. PGI Optimizing Fortran, C and C++ Compilers & Tools. http://www.pgroup.com/index.htm, 2014.

[40] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In Marina C. Chen and Robert Halstead, editors, *PPOPP*, pages 159–168. ACM, 1993.

[41] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. Multi-objective auto-tuning with insieme: Optimization and trade-off analysis for time, energy and resource usage. In *Euro-Par 2014 Parallel Processing*, Lecture Notes in Computer Science, pages –to be published–. Springer Berlin Heidelberg, 2014.

[42] Philipp Gschwandtner, Michael Knobloch, Bernd Mohr, Dirk Pleiter, and Thomas Fahringer. Modeling cpu energy consumption of hpc applications on the ibm power7. In *PDP*, pages 536–543. IEEE, 2014.

[43] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[44] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[45] Intel. Intel C and C++ Compilers. http://software.intel.com/en-us/c-compilers/, 2012.

[46] Suresh Jagannathan and Stephen Weeks. Analyzing stores and references in a parallel symbolic language. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 294–305. ACM, 1994.

[47] Troy A. Johnson, Sang Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in using cetus for source-to-source transformations. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2004.

[48] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 7–18, Piscataway, NJ, USA, 2013. IEEE Press.

[49] Herbert Jordan, Peter Thoman, Juan Jose Durillo Barrionuevo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In Jeffrey K. Hollingsworth, editor, *SC*, page 10. IEEE/ACM, 2012.

[50] Herbert Jordan, Peter Thoman, and Thomas Fahringer. A high-level ir transformation system. In *Euro-Par 2013: Parallel Processing Workshops*, pages 647–656. Springer, 2014.

[51] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 13–22. ACM, 2009.

[52] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[53] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Spire: A sequential to parallel intermediate representation extension. Technical report, Technical Report CRI/A-487, MINES ParisTech, 2012.

[54] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 237–, Washington, DC, USA, 2000. IEEE Computer Society.

[55] Bjoern Knafla and Claudia Leopold. Parallelizing a real-time steering simulation for computer games with openmp. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 219–226. IOS Press, 2007.

[56] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In Allen D. Malony, Mario Nemirovsky, and Samuel P. Midkiff, editors, *ICS*, pages 149–160. ACM, 2013.

[57] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[58] Hugh Leather, Edwin V. Bonilla, and Michael F. P. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO*, pages 81–91. IEEE Computer Society, 2009.

[59] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC*, pages 114–130, 1997.

[60] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC*, pages 197–212, 2008.

[61] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–179, 2003.

[62] Eric Mohr, David A. Kranz, Robert H. Halstead, and Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2, 1991.

[63] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In Donia Scott, editor, *AIMSA*, volume 2443 of *Lecture Notes in Computer Science*, pages 41–50. Springer, 2002.

[64] Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda. *Software Automatic Tuning (From Concepts to State-of-the-Art Results).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2010.

[65] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[66] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[67] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly Media, Inc., 1996.

[68] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

[69] Diego Novillo. Openmp and automatic parallelization in gcc. In *In the Proceedings of the GCC Developers, `http://gcc.gnu.org/projects/gomp/`*, 2006.

[70] Diego Novillo, Ronald C. Unrau, and Jonathan Schaeffer. Concurrent ssa form in the presence of mutual exclusion. In *ICPP*, pages 356–, 1998.

[71] NVidia. Compute unified device architecture (cuda) programming guide, 2007.

[72] Michael O'Boyle and François Bodin. Compiler reduction of synchronisation in shared virtual memory systems. In *ICS*, pages 318–327, 1995.

[73] Stephen Olivier and Jan F. Prins. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, 2010.

[74] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme. In *PACT*, pages 33–42, 2012.

[75] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In Dharma P. Agrawal, editor, *ISCA*, pages 348–354. ACM, 1984.

[76] Zdzisaw Pawlak. Rough sets. *International Journal of Parallel Programming*, 11(5):341–356, 1982.

[77] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. Automatic tuning of mpi runtime parameter settings by using machine learning. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 115–116, New York, NY, USA, 2010. ACM.

[78] Simone Pellegrini, Torsten Hoefler, and Thomas Fahringer. Exact dependence analysis for increased communication overlap. In *EuroMPI*, pages 89–99, 2012.

[79] Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch. Optimizing mpi runtime parameter settings by using machine learning. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 196–206. Springer Berlin Heidelberg, 2009.

[80] Thoman Peter. *Insieme-RS A Compiler-supported Parallel Runtime System*. PhD thesis, University of Innsbruck, 2013.

[81] James L Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[82] E Petit, F Bodin, and R Dolbeau. An hybrid data transfer optimization for gpu. *Compilers for Parallel Computers (CPC2007)*, 2007.

[83] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[84] Antoniu Pop, Albert Cohen, et al. Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers. In *CPC*, 2010.

[85] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[86] L.N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 144–156. IEEE, 2007.

[87] Louis-Noël Pouchet, Peng Zhang, P Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38. ACM, 2013.

[88] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[89] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *PDP*, 2009.

[90] Radu Rugina and Martin C. Rinard. Recursion unrolling for divide and conquer programs. In *Proc. 13th Int. Workshop on Languages and Compilers for Parallel Computing*, LCPC '00, pages 34–48, 2001.

[91] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *ETAPS International Conference on Compiler Construction (CC'12)*, Tallinn, Estonia, March 2012. Springer Verlag.

[92] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference.* MIT press, 1995.

[93] Richard Stallman. Using and porting the gnu compiler collection. *M.I.T. Artificial Intelligence Laboratory*, 2001.

[94] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[95] Rainer Storn and Kenneth Price. Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[96] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[97] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer. Automatic OpenMP loop scheduling: a combined compiler and runtime approach. *OpenMP in a Heterogeneous World*, pages 88–101, 2012.

[98] Peter Thoman. *Insieme-RS: A Compiler-supported Parallel Runtime System.* PhD thesis, University of Innsbruck, 2013.

[99] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In

Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 164–177. Springer Berlin Heidelberg, 2013.

[100] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Compiler multiversioning for automatic task granularity control. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2014.

[101] Peter Thoman, Hans Moritsch, and Thomas Fahringer. Topology-aware openmp process scheduling. In Mitsuhisa Sato, Toshihiro Hanawa, MatthiasS. Müller, BarbaraM. Chapman, and BronisR. Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 96–108. Springer Berlin Heidelberg, 2010.

[102] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[103] Ananta Tiwari and Jeffrey K. Hollingsworth. Online adaptive code generation and tuning. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 879–892. IEEE Computer Society, 2011.

[104] Sven Verdoolaege. barvinok: User guide. *Version 0.23), Electronically available at http://www. kotnet. org/˜ skimo/barvinok*, 2007.

[105] Sven Verdoolaege. *isl*: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[106] Eelco Visser. A survey of rewriting strategies in program transformation systems. *ENTCS*, 57:109–143, 2001.

[107] Eelco Visser. Program transformation with stratego/xt. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

[108] Jürgen Vollmer. Data flow analysis of parallel programs. In *Proceedings of the IFIP WG10. 3 working conference on Parallel architectures and compilation techniques*, pages 168–177. IFIP Working Group on Algol, 1995.

[109] R. Vuduc, J.W. Demmel, and K.A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[110] G.E. Weaver, K.S. McKinley, and C.C. Weems. Score: A compiler representation for heterogeneous systems. In *Proceedings of the 1996 Heterogeneous Computing Workshop*. Citeseer, 1996.

[111] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[112] Anthony Williams. *C++ Concurrency in Action*. Manning; Pearson Education, 2012.

[113] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.W. Liao, C.W. Tseng, M.W. Hall, M.S. Lam, et al. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[114] Jing Yang, Gogul Balakrishnan, Naoto Maeda, Franjo Ivani, Aarti Gupta, Nishant Sinha, Sriram Sankaranarayanan, and Naveen Sharma. Object model construction for inheritance in c++ and its applications to program analysis. In Michael OBoyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 144–164. Springer Berlin Heidelberg, 2012.

[115] Yuan Zhang, Evelyn Duesterwald, and GuangR. Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In Vikram Adve, MaraJess Garzarn, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, volume 5234 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin Heidelberg, 2008.

[116] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.