The Insieme Compiler Frontend: A Clang-based C/C++ Frontend

Master Thesis in Computer Science

by

Bernhard Höckner

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 Master of Science

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

Innsbruck, 20 November 2014

Certificate of authorship/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.

Bernhard Höckner, Innsbruck on the 20 November 2014

Abstract

The Insieme Project provides a research platform for analysing and optimizing parallel programs. It consists of the Insieme Compiler, the Insieme Runtime System and INSPIRE - the *INSieme Parallel Intermediate REpresentation*. INSPIRE is an explicitly parallel intermediate representation, which unifies the representation of parallel APIs and languages such as OpenMP, CILK, MPI, and OpenCL. This intermediate representation is the foundation for analyses and optimizations. Through the Insieme Compiler a programmer is able to tune and optimize such parallel programs.

To support C/C++ based input codes, the Insieme Compiler utilizes the Insieme Frontend, which is capable of parsing and converting C/C++ into INSPIRE. For parsing C/C++, the Insieme Frontend relies on the Clang project. The resulting abstract syntax tree, generated by Clang, is then converted into INSPIRE.

As part of this master thesis an existing (Clang based) C frontend was extended into a frontend capable of handling C and C++ based input codes. This extension required specification and implementation of the conversion process for C++ specific abstract syntax tree nodes from the Clang abstract syntax tree into INSPIRE.

Within this master thesis this conversion process, from the Clang abstract syntax tree into INSPIRE, is documented. This includes an introduction into the abstract syntax tree and its structure, details on how the Insieme Frontend traverses the abstract syntax tree and specifics on the conversion process of abstract syntax tree nodes into INSPIRE.

Furthermore an interception mechanism is implemented in scope of this master thesis. With this interception mechanism the Insieme Frontend is able to support and interface external libaries without converting these libraries into INSPIRE and recompiling them.

Contents

1 Introduction **vii**

1.1 Compilers and their infrastructure viii
1.2 Related Work ix
1.3 Organization x
1.4 Contributions xi

2 Insieme Project **1**

2.1 Insieme Infrastructure Overview 2
2.2 Insieme Compiler Toolset Overview 3
2.3 Insieme Parallel Intermediate Representation 4
2.3.1 Design and Basic Concepts 4
2.3.2 Sequential control flow 6
2.3.3 Extension for Mutable State 7
2.3.4 Extension for Containers 8
2.3.5 Parallel Control Flow 9
2.3.6 Support for object-oriented languages 9
2.4 Insieme Runtime System 10

3 Insieme Frontend **11**

3.1 Overview 11
3.2 Clang AST 14
3.2.1 Overview 14
3.2.2 The Clang Project 15
3.2.3 Setup of the Clang Infrastructure 15
3.2.4 Clang AST nodes 15
3.2.5 Example 19
3.3 Converting Clang AST nodes into INSPIRE 19
3.3.1 Overview 19
3.3.2 Converting `Decl` nodes 22
3.3.3 Converting `Type` nodes 27
3.3.4 Converting `Stmt` nodes 34
3.3.5 Converting `Expr` nodes 38
3.3.6 Examples 49

3.4 IRTranslationUnit . . . 54
3.4.1 Overview . . . 54
3.4.2 Structure and merging of IRTranslationUnit . . . 55
3.4.3 Converting an IRTranslationUnit into an IRProgram . . . 55
3.4.4 Examples . . . 57

4 Supporting Languages and API Constructs 63
4.1 Overview . . . 63
4.2 Plugin System . . . 64
4.2.1 Overview . . . 64
4.2.2 Clang frontend phase . . . 65
4.2.3 Conversion phase . . . 65
4.2.4 Post conversion phase . . . 65
4.2.5 IR phase . . . 66
4.2.6 Pragma handling . . . 66
4.3 Supporting sequential Language Standards . . . 67
4.3.1 Overview . . . 67
4.3.2 C++11 Plugin . . . 67
4.4 Supporting sequential APIs and Third-party Libraries . . . 70
4.4.1 Overview . . . 70
4.4.2 The Interception Plugin . . . 71
4.4.3 The Interceptor . . . 72
4.4.4 Intercepting Third-party libraries . . . 73
4.5 Support for parallel APIs . . . 74
4.5.1 Overview . . . 74
4.5.2 OpenMP . . . 74
4.5.3 CILK . . . 75
4.5.4 MPI . . . 75
4.5.5 OpenCL . . . 75

5 Conclusions and Future Work 77
5.1 Contributions . . . 78
5.2 Future Work . . . 79

List of Figures 81

List of Tables 83

List of Listings 86

Bibliography 87

Chapter 1

Introduction

The development of nowadays hardware architectures started with sequentially processing computations and moved over the last decades towards parallelism. This change was forced by decreasing improvements of the sequential performance due to physical constraints. Hardware vendors applied various techniques and concepts to mitigate the downshift of improvements in sequential computing performance. Some techniques, such as pipelining, branch prediction, super-scalar architectures, are suitable to be optimized by conventional compilers. They require no interaction with the end user, which is in the context of a compiler the software developer. Other techniques, such as single instruction multiple data (SIMD) instructions, or multi-level data and instruction caches are optimized only in simple cases by conventional compilers and still require involvement of the user during tuning and optimizing.

Coarse-grained concepts such as simultaneous multithreading (SMT), symmetric multiprocessing (SMP), or heterogeneous architectures (e.g. GPGPU), rely heavily on the manual interaction with the user to improve the computational performance. Typically these concepts are managed and utilized by the developer via libraries (e.g. Phtread, MPI, OpenCL) or language extensions (e.g. OpenMP, Cilk).

In the exemplary case of a heterogeneous cluster with nodes providing accelerator cards, a developer needs to apply different libraries and frameworks to fully capitalize the available computational power for a software project. For inter-node communication he might apply MPI, for intra-node parallelism OpenMP and Pthreads, and to utilize the accelerator cards, OpenCL would be necessary.

One can easily see that the sheer amount of different libraries and standards makes the process of developing and optimizing a software project a challenging and complex task. Especially the management of interaction between different runtime systems - OpenMP on the intra-node level, MPI on the inter-node level and OpenCL for accelerators - is a time-consuming process during development and may introduce inefficiencies during execution.

To mitigate the complexity of such a software project the developer needs tool support. Such a tool would be for example a compiler and an accompanying runtime system, capable of optimizing the plethora of different APIs and runtime systems.

The goal of the *Insieme Project* is to offer such a compiler and runtime. The *Insieme Compiler* supports multiple input languages and standards such as C/C++, OpenMP, OpenCL, MPI, Cilk. The input code gets translated into a unified, parallel, intermediate representation (INSPIRE) which is the foundation for researching new analyses and transformations tailored towards parallel programs. The generated output programs of the *Insieme Compiler* are targeted towards the *Insieme Runtime System*. Which is able to interact, monitor and dynamically reconfigure hardware, and thus enables and manages parallel execution.

1.1 Compilers and their infrastructure

In general a compiler infrastructure is organized in five major components. The *Intermediate Representation* (IR), accompanied by analysis and transformation tools builds the foundation. Based on the intermediate representation the other components are built: *Frontends* convert the given input code into IR, *Backends* synthesize from the IR the resulting target code. An optional *Runtime System* may offer advanced operations such as resource management or dynamic interpretation or additional compilation support. The *Compiler Driver* is managing the whole compilation process. It organizes the different compilation stages and optimizations conducted on the IR. Further more the Driver offers an interface to the end user - the software developer.

Source-to-Source Compiler A traditional compiler takes source code written in a higher-level language and translates it into a lower-level language. Usually the target language is assembly language or binary, to produce an executable. In other words, the compiler translates the input code from a higher-level of abstraction into a lower-level of abstraction.

In contrast to that, a *source-to-source* compiler takes input source code and translates it again into source code. The level of abstraction is kept on a similar level. This can be leveraged to refactor source code automatically, or to parallelize and annotate source code.

A source-to-source compiler may optimize and transform on a higher-level of abstraction, but is able to utilize the optimizations of a traditional compiler on

the lower-level of abstractions by compiling the generated code with a third-party compiler.

The Insieme Compiler The *Insieme Project* organizes its compiler in a similar way. *INSPIRE* is the intermediate representation, the *Insieme Frontend* converts C/C++ based input codes into INSPIRE. The *Insieme Backend* sythesizes from the *INSPIRE* representation the resulting target code in C/C++. The target code utilizes the *Insieme Runtime System*. Hence the *Insieme Compiler* is a source-to-source compiler.

As implementing a C/C++ parsing frontend is hard, tedious and error-prone, the Insieme Frontend relies on the Clang project [1] to parse C/C++ based input codes. After the input code was parsed, the resulting abstract syntax tree is converted into INSPIRE.

As part of this master thesis an existing C frontend was extended into a frontend capable of handling C and C++ based input codes. This extension required specification and implementation of the conversion of C++ specific abstract syntax tree nodes into INSPIRE. Within this master thesis the conversion from the Clang abstract syntax tree into INSPIRE is documented.

By extending the Insieme Frontend to support C++, the Insieme Compiler is able to offer existing analyses and transformations not only to C input codes but also to C++ input codes. Furthermore the support for C++ builds the foundation to research C++ specific analyses and transformations within the Insieme Compiler.

1.2 Related Work

Among the available conventional compilers and their infrastructures, GCC [2] and LLVM [3] are two large open-source projects which offer a production quality compiler.

- **GCC** - the *GNU Compiler Collection*, offers an open source production-quality compiler. It is widely used in the Linux eco-system.

  GCC uses several intermediate representations with different levels of abstraction: GCC uses different frontends for different languages. These frontends lower their language into a language independent intermediate representation (IR) called *GENERIC*. This GENERIC AST is then lowered further down into *GIMPLE*. GCC uses two kinds of GIMPLE, a higher-level one which is produced after lowering GENERIC by the a middle-end and a low-level GIMPLE which is generated by linearizing control-flow structures used in high-level GIMPLE.

The low-level GIMPLE is then rewritten into single static assignment (SSA) form and translated into RTL (Register Transfer Language) which is the very-low level IR used in the back-end to generate the target code.

In short, GCC uses as intermediate representations GENERIC - generated by the front-ends, high-level and low-level GIMPLE - in the middle-end, and RTL - in the back-end.

- **LLVM** offers an open source production-quality compiler infrastructure. An important goal of the LLVM project is to offer a modular and reusable compiler infrastructure. To achieve this goal LLVM restricts itself to only one RISC-like intermediate representation - LLVM IR. It is based on 31 instructions and enforces SSA form with an infinite amount of registers.

  A variety of frontends lower their input code into LLVM IR and after running optimizations and transformation, a variety of backends generates the target code.

Source-to-source Compilers Two exemplary production-quality source-to-source compilers are the Clang project [1] and the Rose compiler project [4]. Each project uses a rather rich and detailed abstract syntax tree (AST) to capture all the information of the given input code. This includes comments and code formatting details to accurately reproduce the input code if necessary. This can be leveraged for static analyzing or refactoring tools.

- **Clang** is the C language family frontend for the LLVM project. It parses the input code into the Clang AST. Besides generating LLVM IR from the Clang AST as target code it can also be used as source-to-source compiler and generate code with a similar level of abstraction as the input code.

- **ROSE** is a research-oriented compiler project. It uses a commercial, external frontend, from EDG [5], and has its own abstract syntax tree - Sage III also known as the ROSE IR. The goal of the ROSE project is in providing an infrastructure for source-to-source and analysis tools.

1.3 Organization

This master thesis is structured into three chapters. In Chapter 2 we cover the Insieme infrastructure consisting of the *Insieme Compiler* and the *Insieme Runtime System*. We introduce *INSPIRE* (*INSieme Parallel Intermediate REpresentation*), the intermediate representation building the foundation of the *Insieme Compiler* and its components.

The *Insieme Frontend* is covered in Chapter 3, it covers how the C/C++ input code is parsed into an abstract syntax tree, and converted into *INSPIRE*.

In Chapter 4 we detail how the conversion process for C/C++ can be altered and adjusted to support further C/C++ based input languages. This includes OpenMP, CILK, MPI and OpenCL. We also elaborate how the conversion of additional C/C++ language standards such as C++11 can be supported by the *Insieme Frontend*.

1.4 Contributions

The authors contributions were the extension of the existing C frontend based on the Clang infrastructure to support C++. In the course of this work the conversion of several C++ specific Clang AST nodes was specified and the specification implemented.

To enable support for third-party libraries within the Insieme infrastructures the author implemented an intercepting mechanism. This intercepting mechanism offers means to model code in IR interfacing third-party libraries. The *Interceptor* utility and an accompanying plugin were created.

Another major contribution is this document, which provides documentation of the Insieme Frontend itself.

Chapter 2

Insieme Project

In this chapter we give a short introduction of the *Insieme Compiler Project* of the *Parallel and Distributed Systems Group* at the *University of Innsbruck*. The mission statement [6] for the Insieme Project on the Insieme Project homepage [7] states:

> The main goal of the *Insieme project* of the *University of Innsbruck* is to research ways of automatically optimizing parallel programs for homogeneous and heterogeneous multi-core architectures and to provide a source-to-source compiler that offers such capabilities to the user.

The following features are stated as essential in achieving this goal [6]:

- support for multiple programming languages and paradigms such as C, Cilk, OpenMP, OpenCL and MPI (C++ support is under development)
- multi-objective optimization techniques supporting objectives such as execution time, energy consumption, resource usage efficiency and computing costs
- the Insieme Runtime that provides an abstract interface to the hardware infrastructure, offering online code tuning and steering, dynamic reconfiguration of hardware resources and monitoring of the application's performance
- an input code independent Intermediate Representation (INSPIRE) for developing new compiler techniques to optimize parallel programs

These essential features are provided and implemented by the Insieme infrastructure consisting of the Insieme Compiler which is using the INSieme Parallel Intermediate REpresentation and the Insieme Runtime System.

The support for "*multiple programming lanuages and paradigms*" is provided by the Insieme Compiler. The Insieme Compiler is a source-to-source compiler,

which takes parallel input codes based on C and C++ and produces output code to utilize the Insieme Runtime System. Currently the supported parallel input languages and standards are OpenMP, OpenCL, MPI and Cilk.

To enable extensive analyses and transformations in the Insieme Compiler, the INSieme Parallel Intermediate REpresentation (INSPIRE) is used as intermediate representation (IR). The design goal of INSPIRE was to represent all the different parallel input languages (OpenMP, OpenCL, MPI, Cilk) in an *unified* intermediate representation. Further tools are provided by the Insieme Compiler to perform and implement analyses and manipulations on INSPIRE.

The Insieme Runtime System has been designed to run parallel programs represented in INSPIRE and translated by the Insieme Compiler. The Insieme Runtime System offers means to monitor and interact with hardware, and enables and manages parallel execution. These functions are utilized by the code generated by the Insieme Compiler. The means of control offered by the Insieme Runtime System are leveraged to optimize the various objectives (i.e. execution time, energy consumption, efficient resource usage, computing costs) stated.

2.1 Insieme Infrastructure Overview

The Insieme infrastructure consists of the Insieme Compiler, the Insieme Runtime System and the INSieme Parallel Intermediate REpresentation (INSPIRE). The general structure of the infrastructure is fixed, e.g. the Insieme Frontend is responsible for translating the input code into INSPIRE, the Insieme Analyses & Transformations are utilized for optimizations, the Insieme Backend produces the output code to be compiled with the backend compiler. Currently there are two backends, one generating sequential code, and the other one generating code utilizing the Insieme Runtime System. So the choice of the Insieme Backend, influences the setup in respect to the Insieme Runtime System.

A typical setup of these components can be found in Figure 2.1. It is possible to use the Insieme Compiler to translate the given input code from the C/C++ based input language (OpenMP, OpenCL, MPI, Cilk) into its INSPIRE representation, run analyses and transformations on the INSPIRE representation and produce output code to utilize the Insieme Runtime System. The produced output code is compiled by the Backend Compiler, which is a secondary compiler - usually GCC - into an executable binary.

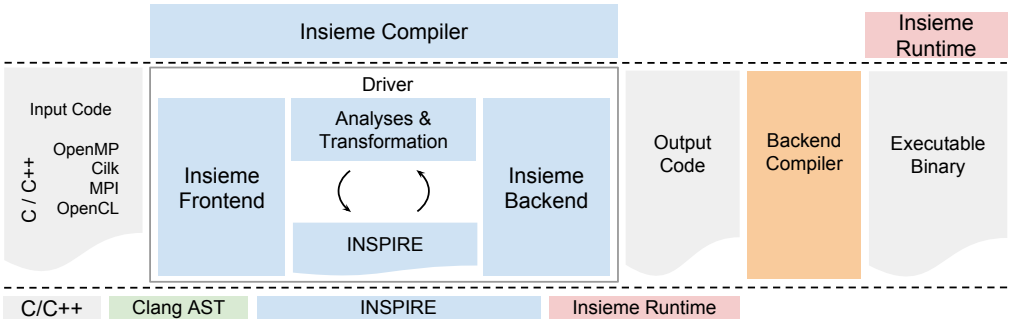

Figure 2.1: Typical setup of the Insieme infrastructure

2.2 Insieme Compiler Toolset Overview

The Insieme Compiler itself is organized in different components: *Frontend*, *Backend*, *Analyses/Transformations*. A *Driver* is organizing and managing the different stages of the compiler.

Frontend The Frontend is responsible for parsing the input code into an abstract syntax tree (AST) and translating the AST into the semantically equivalent INSPIRE representation. To avoid the tedious implementation of a C family languages frontend, an external software was used. Clang [1] is the frontend project used by the LLVM compiler project [3]. It combines a solid translation of C/C++ languages into an AST with a powerful set of tools to manipulate the AST. The Insieme Frontend is strongly influenced by the tools provided by the Clang project.

Analyses/Transformations Offers the tools and means to implement and research various analyses and transformations on the INSPIRE representation.

Backend From the INSPIRE representation the output code is created. The Backend is capable of generating output code to utilize the Insieme Runtime System or a sequential version.

(Compiler) Driver A Driver can be seen as the main interface for a end-user (e.g. in the context of a compiler, a software developer) to the compiler's tools. It sets up the environment and conducts the sequence of the different stages of the Insieme Compiler. As pictured in Figure 2.1, the Driver orchestrates the Frontend, Analyses, Transformation and Backend. Depending on the intended use, there can be different drivers for one compiler.

One example for a driver is the *insiemecc* driver for the Insieme Compiler. It can be used as a replacement for GCC or any other C/C++ compiler. It collects all the inputs (e.g. input files, include paths, library locations), and options (e.g. language standard for the input code, support for certain parallel language APIs) needed for the compilation process.

2.3 Insieme Parallel Intermediate Representation

In the following section we present a short overview of the INSieme Parallel Intermediate REpresentation (INSPIRE) according to Jordan et al. [8]. First we present the design principles and basic concepts of INSPIRE, followed by sections on the sequential and parallel control flow, and support for object-oriented languages. This section is only meant to introduce INSPIRE, its design and usage. Further we present an overview of C++ features and their representation in INSPIRE. For a more complete introduction see Jordan et al. [8]. More details and reasoning on the design of INSPIRE can be found in Herbert Jordan's thesis [9]. The content of this section can be seen as an introductory summary of Jordan et al. [8] and Herbert Jordan's thesis [9].

2.3.1 Design and Basic Concepts

The INSieme Parallel Intermediate REpresentation (INSPIRE) is the intermediate representation used by the Insieme Compiler Infrastructure to represent programs and run analyses and transformations to achieve optimizations. The intermediate representation is defined over an "extensible, high-level intermediate language comprising explicit parallel constructs to unify the representation of heterogeneous parallel codes" [8]. As stated in Jordan et al. [8] and Herbert Jordan's thesis [9] an intermediate representation should be *expressive*, *analyzable*, *transformable* and *extensible*. Some of these criterias conflict with each other. Thus for INSPIRE were five main design principles derived:

Explicit - INSPIRE shall be explicit in all important concepts, e.g. parallelism and memory management, to simplify analyses and enable transformations.

Unified - INSPIRE shall represent different input language constructs with the same meaning in the same constructs. This results in one formalism to represent different parallel input languages such as OpenMP, Cilk, MPI, OpenCL. This means that if the input languages use different constructs to express the same meaning, then the resulting INSPIRE representation should be the same for all input languages. The different input languages

vary in the way they are implemented, for example OpenMP uses compiler directives (pragmas) and a library, Cilk is a language extension to C/C++ and implements their parallelism by using additional keywords, MPI uses libraries to implement the parallelism, and OpenCL uses libraries and Just-In-Time compilation. INSPIRE defines a parallel model which is capable to cover all the input languages and their formalism for parallel control flow. All these varying approaches need to be recognized by the Frontend and translated into the corresponding INSPIRE constructs.

Simple - INSPIRE shall have constructs with a precise and non-overloaded interpretation.

Modular - INSPIRE shall offer a fixed *language core* and means to define extensions. These extensions are defined within the language and not its implementation.

Compact - INSPIRE shall have as few constructs as necessary.

Based on these design principles, INSPIRE was designed with influences from functional programming languages and formal specification languages. From functional programming languages the property of functions are first-class citizens, thus powerful functional composition is taken and from formal specification languages the concept of abstract types and operators over such abstract types is inherited. The type system offers type variables to enable generic functions and abstract types. With these abstract types INSPIRE represents primitive types, basic data structures and interfaces for external libaries. Furthermore additional abstract types can be added without modifications on the language itself.

INSPIRE is defined in two components: the *language core* and *extensions*. The *language core* covers a fixed set of primitives: types, expressions and statement constructs, also it specifies the deduction of types, composition and evaluation-order of expressions, statement processing and variable scope.

Based on the *language core*, *extensions* can be defined. Either by using abstract constructs, or - preferably - by composing already defined constructs into *derived constructs*. When abstract constructs are used, the utilities handling INSPIRE (e.g. Analysis, Transformations, the Backend), need to interpret them correctly.

The data structure of INSPIRE is again influenced by formal and functional languages and their self-contained structure. Logically INSPIRE is a tree structure without cross or back edges. The overall structure of a resulting INSPIRE tree is not reflecting the organization of the input source files but rather the

execution of the code. Hence the root node of the tree represents the execution of a code fragment, typically the main function of a program or separate libary functions. In addition an INSPIRE tree stores the expression representing the target function for function calls right at the call site. This results in removing the distinction made by C/C++ between translation units or header and implementation files. Due to this approach INSPIRE allows to facilitate context sensitive or whole-program optimizations. As this self-contained structure would require huge memory due to the duplication of functions and types the physical representation of the logical INSPIRE tree is a DAG with node sharing.

INSPIRE is only focused on the semantic aspect of a program. To provide additional information (e.g. analysis results, loop-scheduling policies) every IR node can be annotated with generic information. These annotations can forward meta-information to the various stages of the compiler for example the analysis results to the Transformation stage, or loop-scheduling information to the Runtime System.

2.3.2 Sequential control flow

Here we list the constructs used within INSPIRE to represent sequential control flow, as presented in Jordan et al. [8] and Herbert Jordan's thesis [9].

Types

To define types in INSPIRE the following constructors are offered. The actual definitions of these constructors are presented in Jordan et al. [8]. Primitive types like boolean (*bool*), signed 4-byte integer ($int\langle 4\rangle$), or double ($real\langle 8\rangle$) are defined using *parametrized abstract types*. Besides such primitive types, basic data structures are also defined with parametrized abstract types. For example dynamically sized array of booleans - $array\langle bool\rangle$ - or statically sized vector of integers - $vector\langle int\langle 4\rangle\rangle$. Then there are type constructors to construct *struct*, *union*, *function*, and *closure* types. To define generic types *type variables* can be used. Typically we use greek letters (α, β) or *'a, 'b* as identifiers for type variables. Furthermore there is a constructor to define *recursive types*.

Expressions

To model data and control flow the constructors for *variables*, *literals* and *call* expressions are provided. There are four expressions to construct *struct*, *union*, *function*, *closure* values. A literal is a typed constant in INSPIRE. This can be utilized to introduce an abstract function or operator by using a literal with a function type. When generic functions are invoked using a *call* expression, the

type of the generic function is instantiated according to the arguments. For all expressions a type is deduced with inductive type deduction rules.

Statement

In INSPIRE every expression is a statement. Furthermore there are eight statements typically found in imperative languages: *compound*, *variable declaration*, *if*, *while*, *for*, *return*, *break*, *continue*.

2.3.3 Extension for Mutable State

Until now we only described and used the *language core* and its *pure* features i.e. side-effect-free features, which imply immutable data. As the input language (i.e. C/C++) needs mutable state we need a way to model it in INSPIRE. The language core is extended by using the abstract generic types and generic functions with the reference type: $ref\langle\alpha\rangle$. Here α is a type variable. A value of this type represents a reference to a memory location containing a value of type α. Such a memory location represents a C/C++ variable of the given type. To operate on references several operators are defined. An incomplete list of the available operators is given in Table 2.1. Additionally to the operators shown in the table comparison operators are provided.

Operator	Type	Description
ref.var	$(type\langle\alpha\rangle) \rightarrow ref\langle\alpha\rangle$	allocation of stack memory
ref.new	$(type\langle\alpha\rangle) \rightarrow ref\langle\alpha\rangle$	allocation of heap memory
ref.delete	$(ref\langle\alpha\rangle) \rightarrow unit$	deallocation of heap memory
ref.deref	$(ref\langle\alpha\rangle) \rightarrow \alpha$	read value from memory location
ref.assign	$(ref\langle\alpha\rangle, \alpha) \rightarrow unit$	update/assign value in memory location
ref.reinterpret	$(ref\langle\alpha\rangle, type\langle\beta\rangle) \rightarrow ref\langle\beta\rangle$	casting memory location of type α to type β

Table 2.1: Operators of ref extension

Only `ref.var` and `ref.new` operators can allocate memory, either bound to the current scope's stack or to the heap. The $type\langle\alpha\rangle$ is a generic type used to specify the type of the memory location to allocate. The `ref.assign` operator is the only operator able to mutate the state of a memory location.

With references the requirement to differ between r-values and l-value is removed as these two cases can be distinguished by the type. In C the `bool` type

can be a mutable memory location (an l-value) as well as the result from an operation (an r-value). In INSPIRE the first would be an expression of type $ref\langle bool\rangle$ while the second would be a value of type *bool*.

To model a C/C++ variable of a constant type (e.g. `const bool`) we set the kind of the *ref* type to *source* thus it is then called a *src* type. Such a *src* reference models a read-only memory location. Furthermore the kind of *ref* type can be set to *sink*, which is than a write-only *ref* and is called a *sink*. For the *sink* type there is no equivalent in C/C++.

2.3.4 Extension for Containers

Another use of the abstract generic types is the support for container types like arrays, lists, and vectors. In INSPIRE the abstract generic type $array\langle\alpha\rangle$ represents a dynamically sized array with elements of type α. Besides the dynamically sized array INSPIRE has an extension to model statically sized vectors modeled as the abstract generic type $vector\langle\alpha, \#elem\rangle$ which represents a statically sized vector of size $\#elem$ with elements of type α. Accompaning the generic types are operators to manipulate and access these containers.

Operator	Type	Description
array.create	$(type\langle\alpha\rangle, uint\langle 8\rangle) \rightarrow array\langle\alpha\rangle$	creates an (immutable) array with the given number of undefined elements
array.elem	$(array\langle\alpha\rangle, uint\langle 8\rangle) \rightarrow \alpha$	obtains the value at a given index position
array.ref.elem	$(ref\langle array\langle\alpha\rangle\rangle, uint\langle 8\rangle) \rightarrow ref\langle\alpha\rangle$	accessing an element of an mutable array

Table 2.2: Operators on *array* container

In general the result of the *array.create* operator will be assigned to an reference refering to a memory location holding an element of array type, i.e. a variable of type $ref\langle array\langle\alpha\rangle\rangle$. Access to an element of such an mutable array the *array.ref.elem* operator is used, this obtains a reference to the adressed element which can than be read with the *ref.deref* operator or updated with the *ref.assign* operator.

Similar operators as for the *array* type exist for the *vector* type.

2.3.5 Parallel Control Flow

To model the parallel control flow INSPIRE defines a *parallel model* based on recursively nested thread groups, and explicit parallel constructs like *jobs*, *spawn*, *merge*, *pfor*. With this parallel model INSPIRE is able to represent the various input languages (OpenMP, Cilk, MPI, OpenCL) with one unified formalism. More details on the parallel model can be found in Herbert Jordan's thesis [9] and Peter Thoman's thesis [10].

2.3.6 Support for object-oriented languages

INSPIRE provides additional constructs to represent object-oriented features like inheritance relationships for types, and means to represent constructors, destructors and member functions.

The design objective to be *explicit* was dropped for several features found in C++ as it turned out to be to ambitious to represent every implicit semantical detail of C++ explicitly in INSPIRE. Some of the features which were kept implicit e.g. implicit life-cycle management of objects – no explicit triggering of the destructor of an variable at the end of its scope, implicit copy constructors for passing arguments by value.

Then again several of the implicit features of C++ are made explicit e.g. constructor invocations, virtual functions, implicit type conversions, overloading of operators, memory management.

Several features of C++ were omitted in INSPIRE as they were found to be not relevant for the semantics of the program, e.g. access modifiers (private/protected/public) – everything is public, names and namespaces, translation units, split up of header/implementation. These features exist either for the convenience of the user or for increasing the users productivity.

An important reason for the success and wide application of C++ is its support for libraries and their smooth integration into the users code. An important example for such a library is the STL providing the user with standard implementations for e.g. containers, algorithms and IO. Thus for INSPIRE it was an important goal to keep the ability to interface C++ libraries provided by a third-party without the need to translate these libraries into INSPIRE.

Instead of modelling the structure of a program, i.e. how the program is organized into translation units, and header or implementation files, INSPIRE models the actual execution. This results in the fact that only code which is actually executed is represented in an INSPIRE representation of an program.

In some cases a C++ class provides member functions which are not invoked explicitly in the user code. One example are copy constructors – which are invoked implicitly when passing or returning a value to a function but are rarely

invoked explicitly by the users code. Another example is the requirement of some containers that the object it stores provides certain comparison operators to offer sorting utilities. Again these operators might not be invoked explicitly from the users code but inside the third-party library providing the container. As these member functions might not be invoked explicitly, they might not be covered by the INSPIRE representation of the code, and therefore lost. To solve the issue with such possibly implicit features and keep the compatibility with such libraries and cover all the member functions, the *ClassMetaInfo* was introduced. It collects all member functions (e.g. constructors, destructor, (virtual) member functions, operators) and attaches them as an annotation, to the class type. The Backend is responsible to generate output code for the functions present in the *ClassMetaInfo*.

2.4 Insieme Runtime System

The Insieme Runtime System is the main target of code generated by the Insieme Compiler, and an essential part of the Insieme Infrastructure, hence it is introduce here. For a complete introduction and greater details on the Insieme Runtime System see the dissertation of Peter Thoman [10].

The parallel programs generated by the Insieme Compiler and run with the Insieme Runtime System, allows those programs to interact with and manage the hardware they are run on, and enable and manage their parallel execution. With the Insieme Runtime System the executed program is able to leverage parallelism on coarse and fine-grained level. The generated code maps the INSPIRE parallel model to the program model provided by the Insieme Runtime System.

Chapter 3

Insieme Frontend

3.1 Overview

This chapter details the conversion process for C/C++ input codes into the INSPIRE representation. The result of this conversion process is the intermediate representation (IR) of the given input codes in INSPIRE. Throughout this master thesis the term IR is used for the intermediate representation of the input codes in INSPIRE.

In Figure 3.1 we illustrate the overall architecture of the Insieme Compiler with emphasis on the Insieme Frontend. After the Insieme Frontend completed its conversion process, the resulting IR is stored in an IRProgram - a container storing the entry points of the input code (e.g. the `main` function) and their corresponding IR - the INSPIRE tree. In the Insieme Compiler, this IRProgram is passed on to the subsequent stages of Analyses/Transformations and finally the Backend is responsible of generating output C/C++ code.

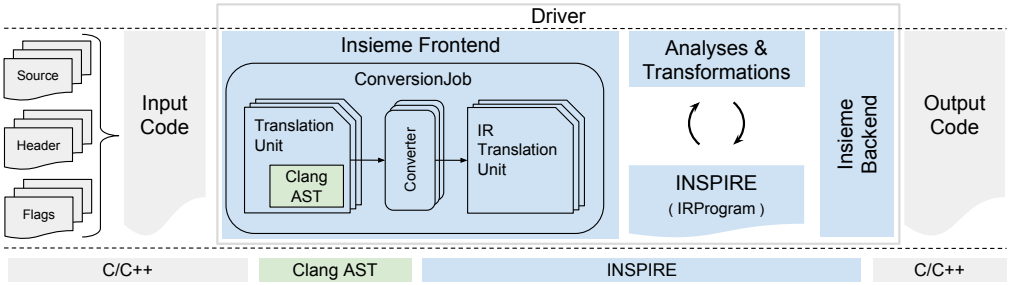

Figure 3.1: Overview of the Insieme Compiler

We start by outlining the whole conversion process and the involved components as pictured in Figure 3.2. Following this overview we detail the main components involved in the conversion: Section 3.2 covers the Clang AST, Section 3.3 details the Converter and in Section 3.4 the IRTranslationUnit is explained.

The Driver is the interface to the user and responsible for preparing and managing the different stages of the Insieme Compiler. After the Driver collected

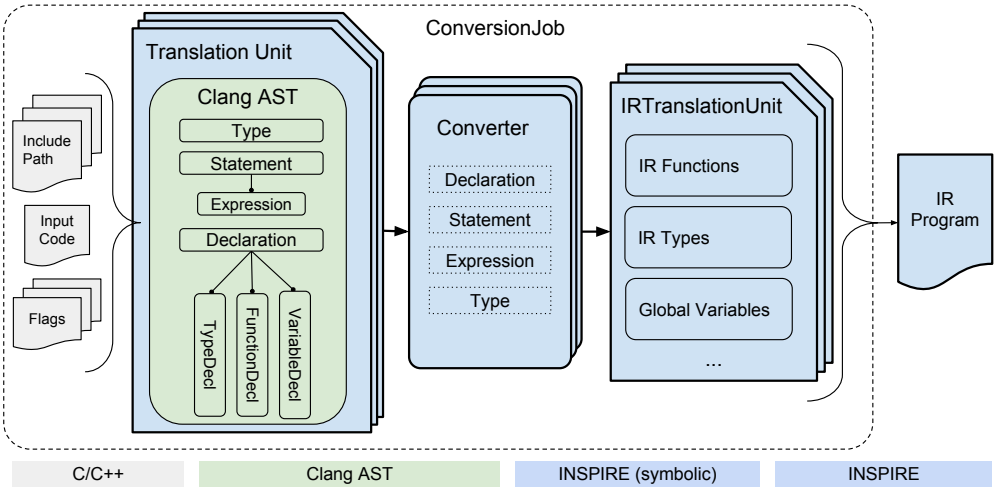

Figure 3.2: Processing a ConversionJob

all the user input, a ConversionJob is set-up with all the inputs relevant for parsing and converting the given input code. This includes amongst other: source code files, include directories, and flags specifying the input language (i.e. C/C++, OpenMP, Cilk, MPI, or OpenCL). A ConversionJob stores all these user inputs and manages the conversion from C/C++ into INSPIRE.

Source files and translation units First a short overview on the structure of C/C++ programs: a C/C++ program is structured in source files and header files. The header files typically declare the interface to an external library or a component of the program. These header files are referenced by the source file with the `#include` directive of the preprocessor. The preprocessor knows how to search the header file in the include directories specified by the user. After preprocessing a translation unit is formed that contains the contents of the source file and all its included headers. One such translation unit can then be compiled into one object file, resulting in a library, or an executable. Simply put, a translation unit is a source file and its included header files.

For preprocessing and parsing, the Insieme Frontend uses the Clang Project [1]. Thus we leverage the abstract syntax tree used by the Clang project: the Clang AST. To support the different parallel input languages (e.g. OpenMP, Cilk, MPI, OpenCL) we need Clang to be able to preprocess and parse them. Now to the actual conversion process from C/C++ to INSPIRE. This process can be split into two steps:

C/C++ to Clang AST The first conversion step, from C/C++ source code to the Clang AST, is handled in the Insieme Frontend by the Translatio-

nUnit component. For each C/C++ translation unit there is one single TranslationUnit instance. Such a TranslationUnit sets up the Clang infrastructure needed to preprocess and parse the given C/C++ translation unit into its Clang AST.

Clang AST to INSPIRE The second conversion step, from the Clang AST to INSPIRE, is handled by the Converter components. The basic nodes of the Clang AST describe declarations, statements, expressions, and types. These basic nodes are handled by separate Converters. To ease the transition from the translation unit oriented approach used by C/C++ and Clang AST to the execution oriented approach used by INSPIRE we introduced the IRTranslationUnit component. An IRTranslationUnit stores a symbolic form of INSPIRE. This symbolic IR relaxes the self-contained property of the logical INSPIRE by introducing symbols for types (i.e. an abstract type) and functions (i.e. abstract function) to be used instead of the actual implementation of these types and functions. For example whenever there is a call to a function instead of the implementation (i.e. an INSPIRE expression) we use the symbol. This symbolic form of INSPIRE is only used in the Frontend and assists with the resolution of recursive types and functions. From the symbolic IR stored in the IRTranslationUnit the final IRProgram is resolved.

As a C/C++ program is usually composed by multiple translation units, the ConversionJob converts every single translation unit into its corresponding IRTranslationUnit. At the end of the conversion process all the IRTranslationUnits are merged into one single IRTranslationUnit. That single IRTranslationUnit is then resolved into the IRProgram.

Program representations and their structure A different view on this process gives Figure 3.3, illustrating the different program representations and their structure. Throughout the conversion process, C/C++ source code, Clang AST, symbolic INSPIRE, and INSPIRE are used as program representation. In Figure 3.3 (a) we see two separate source files, after preprocessing, i.e. translation units. These two translation units are parsed into two Clang ASTs in Figure 3.3 (b) and converted into two separate IRTranslationUnits in Figure 3.3 (c). After merging the IRTranslationUnits, they are resolved into the IRProgram. Figure 3.3 (d) shows the logical structure, and Figure 3.3 (e) shows the physical structure of INSPIRE.

Details on the structure of Clang AST are presented in Section 3.2. The conversion steps for the Clang AST nodes into the symbolic form of INSPIRE are

given in Section 3.3. The logical and physical structure of INSPIRE were introduced in Section 2.3. In Section 3.4 we detail the structure of IRTranslationUnit and how to resolve the symbolic INSPIRE into the logical INSPIRE.

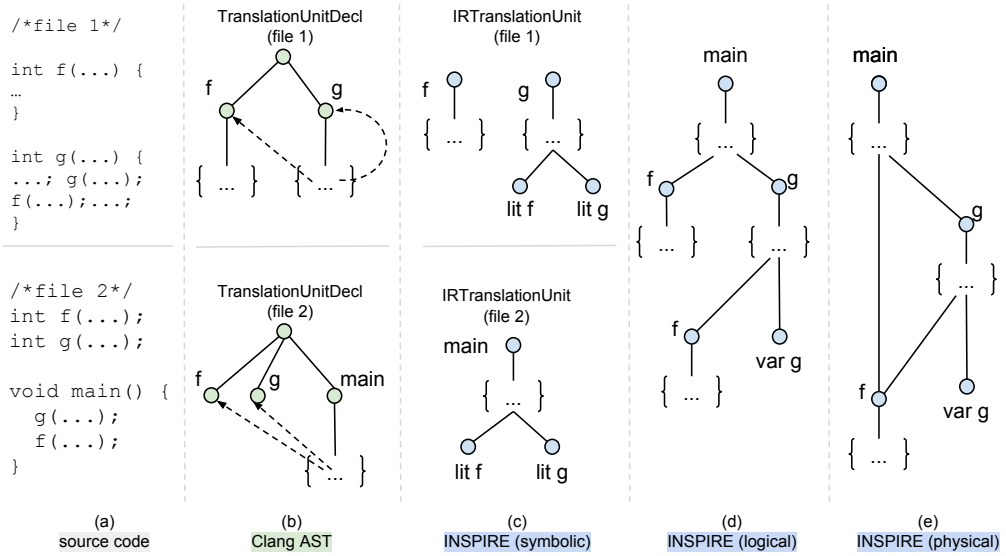

Figure 3.3: Structures of different representations

To support the various parallel input languages the conversion process has to be adapted to the specific needs. We provide a plugin system which is able to alter the conversion in critical points like the set-up of the Clang infrastructure itself or in the Converters of the Clang AST. The implementation of this plugin system is detailed in the master thesis of Stefan Moosbrugger [11].

Details on the support and conversion process for the various sequential and parallel input languages and APIs are discussed in Chapter 4.

3.2 Clang AST

3.2.1 Overview

For preprocessing and parsing the given C/C++ input codes we leverage the Clang project [1] and its abstract syntax tree – the Clang AST. This section covers the conversion of one C/C++ input translation unit into its corresponding Clang AST. First we give a short introduction of the Clang Project, then a short overview of the setup of the Clang infrastructure in a TranslationUnit. The main part is an introduction of the Clang AST and its main nodes describing declarations, statements, expression and types. The goal of this section is not to explain the Clang AST in full detail but to give a basic understanding of

the nodes and how the Clang AST is structured. This basic understanding is needed for Section 3.3, describing the conversion from the Clang AST to INSPIRE. The Clang project provides a more complete introduction [12] and a extensive documentation [13] of the Clang AST and its many different kinds of nodes.

3.2.2 The Clang Project

The goal of the Clang project is to build a frontend for the LLVM Project[3] supporting the C language family. As stated in the Clang user manual [14], the C language family includes C/C++ and Objective-C/C++ and various dialects. The Clang Project aims to provide a library-based architecture to be utilized in diverse utilities and applications which offer fast compilation and low memory consumption.

Currently the Insieme Project only targets C/C++ input codes therefore the Insieme Frontend is only utilizing the C/C++ nodes of the Clang AST.

3.2.3 Setup of the Clang Infrastructure

For each translation unit of the input code we setup a TranslationUnit object, which is responsible for parsing the given translation unit into the Clang AST. The TranslationUnit object sets up the Clang infrastructure with the given translation unit and configures it with the additionally user-provided options. These options include for example the specific input language (e.g. C or C++), the language standard, include directories, or preprocessor macro definitions.

When the setup is finished, the input code is preprocessed and parsed, resulting in a Clang AST.

3.2.4 Clang AST nodes

The nodes of the Clang AST are organized in multiple class hierarchies, which do not share a common base class. The basic nodes of the Clang AST are `Decl` (i.e. a declaration), `Stmt` (i.e. a statement), `Expr` (i.e. a expression) and `Type` (i.e. a type). These basic nodes are the base classes for rather large class hierarchies. Beside these basic nodes, and their rather large hierarchies, plenty of nodes exist which are part of no hierarchy or only a small hierarchy. Use cases of such smaller hierarchies would be modelling inheritance relation of classes or modelling templates. Nodes which are specifically used to support a certain language (e.g. C, C++, Objective-C/C++) have a prefix to indicate this. Nodes only used for C++ use `CXX` as prefix. In the following, we present an

incomplete selection of nodes. For a complete list of the available nodes check the Clang AST documentation [13].

Declarations

Declarations are modelled in the `Decl` node. Some `Decl` nodes inherit from `DeclContext` – they act as declaration context for other declarations, or in other words: declarations which can contain other declarations. Examples for such nodes would be the `TranslationUnitDecl` node or the `FunctionDecl` node. An example for a declaration which is not a declaration context for other declarations would be a variable declaration.

As the Clang AST has numerous declaration nodes, inheriting from the `Decl` node, we present only an incomplete list with the more relevant nodes for the Insieme Frontend:

- `TranslationUnitDecl` is the main entry into the Clang AST of one C/C++ translation unit (i.e. a preprocessed input file with all its headers), from there we can traverse the whole Clang AST. `TranslationUnitDecl` is a `DeclContext` as it can contain further declarations like: function declarations, type declarations, or declarations of global variables.

- `FunctionDecl` nodes model function declarations. The child nodes of a `FunctionDecl` node model the various aspects of a function declaration or function definition like the function parameters, or the function body. From `FunctionDecl` the nodes for C++ methods (`CXXMethodDecl`), constructors (`CXXConstructorDecl`) and destructors (`CXXDestructorDecl`) are derived.

- `VarDecl` nodes model variable declarations and their requirements for memory allocation. The node modelling function parameters (`ParamVarDecl`) inherits from `VarDecl`.

- `TypeDecl` nodes model the declaration and layout of (user-defined) types. The nodes for class/struct types (`CXXRecordDecl`) or enumeration types (`EnumDecl`) as well as typedef's (`TypedefDecl`) are derived from `TypeDecl`.

Statements

Statements are modelled in the `Stmt` node. Every `Stmt` node provides means to access all it's children - substatements or subexpressions. In general, statement nodes are used for both C and C++. Following is an incomplete list of the available `Stmt` nodes:

`CompoundStmt` - a block of statements: {...}. The statements inside the block are organized as a list of `Stmt` nodes.

`IfStmt` - an `if` statement, child nodes are for example the `then` and `else` branch and the condition expression.

`ForStmt` - a `for` loop, child nodes are for example the initialization statement, the condition, the increment expression and of course the loop body

`WhileStmt` - a `while` loop, child nodes are for example the condition, and of course the loop body

`DoStmt` - a `do-while` loop, similar to the `WhileStmt` node. The child nodes are for example the condition, and of course the loop body

`ReturnStmt` - a `return` statement

Further `Stmt` nodes would be the `ContinueStmt`, `BreakStmt`, `GotoStmt`, `SwitchStmt`. There is only a limited number of C++ specific `Stmt` nodes such as:

`CXXTryStmt` - a `try` block

`CXXCatchStmt` - a `catch` block

Expressions

Expressions are modelled in the `Expr` node, which is a subclass of `Stmt`. This means a `Expr` node can be used everywhere a `Stmt` node could be used.

In general expression nodes are used for both C and C++.

`CallExpr` - a call to a function. Its child nodes describe the callee and its arguments.

`DeclRefExpr` - a `Expr` node referring to a declaration like a function or variable, in other words the usage of a previously declared C/C++ symbol. For example the callee of a function call is typically a `DeclRefExpr`.

`UnaryOperator` - the various unary operators and the expression it is applied on as a child

`BinaryOperator` - the various binary operators and the expressions, left-hand-side and right-hand-side, they are applied to, as children

There are several C++ specific `Expr` nodes:

`CXXMemberCallExpr` - inherited from the `CallExpr` node, models a call to a C++ member function. Child nodes are the callee and its arguments, as well as the `this`-object the member function is called on.

`CXXOperatorCallExpr` - inherited from the `CallExpr` node, models a call to overloaded operator. For overloaded operators written as member function, is the `this`-object also a child node.

`CXXConstructExpr` - a constructor call. Its child nodes describe the callee and its arguments.

`CXXThisExpr` - the `this` keyword. Represents the `this`-object.

`CXXNewExpr/CXXDeleteExpr` - models `new/new[]` and `delete/delete[]` operators for memory allocation and deallocation. For class types it has child nodes to the constructor and destructor.

`CXXThrowExpr` - models the `throw` expression for exceptions

Types

Types are modelled in the `Type` node. In general type nodes are used for both C and C++ like:

`BuiltinType` - builtin types like integers (e.g. `short`, `int`), floating point (`double`, `float`), character `char`, `void` or boolean (`bool`).

`PointerType` - pointer types, like `<type>*`. Where `<type>` is a child node of the `PointerType` node of the type the pointer points to.

`EnumType` - enumeration types.

`RecordType` - unions, classes and struct types.

`FunctionType` - function types, their return type and possibly the parameter types.

There are several C++ specific type nodes. Some will only be used if a specific language standard is used:

`ReferenceType` - models the C++ reference type

`AutoType` - models the C++11 `auto` type

`MemberPointerType` - models a pointer to a member of a C++ class or struct, can be a data member or a function member

`Template...Type` - several nodes to model template types

Type Qualifiers Additionally to the types, C/C++ offers qualifiers (e.g. `const`, `volatile`) for types. These qualifiers influence details on the memory allocation or the allowed useage for a particular value of that qualified type. For reasons of efficiency, Clang uses for types with a type-qualifier (e.g. `const`, `volatile`) not individual nodes but uses an additional node only used for modelling the qualifiers: `QualType`. The actual type is a child of the `QualType` node. This reduces the number of type nodes as their is no need to represent types like `int`, `const int`, `volatile int`, `const volatile int` in their own nodes but by using a `QualType` node and the underlying type as one of its children.

3.2.5 Example

A simple way to transform the Clang AST into a textual representation, is by using the Clang compiler itself. With this command one gets the Clang AST of the given input file: `clang -Xclang -ast-dump <input file>`.

The input translation unit shown in Listing 3.1 consists of two functions i.e. `func1` and `func2`. This results in the Clang AST given in Listing 3.2, consisting of an `TranslationUnitDecl` and two `FunctionDecl` nodes for the functions `func1` and `func2`. Each of these functions bodies are represented as a `CompoundStmt` and other `Stmt` nodes representing the statements inside the functions bodies.

```
1 void func1() {
2   int x;
3   int y = 1;
4   x = y+2;
5 }
6
7 void func2() {
8   func1();
9 }
```

Listing 3.1: C function with a Call

3.3 Converting Clang AST nodes into INSPIRE

3.3.1 Overview

This section covers the conversion of the Clang AST, and its individual nodes, into IR. The results of this conversion are stored in an IRTranslationUnit.

```
TranslationUnitDecl 0xb7390 <<invalid sloc>>
... Clang internal declaraions ..
|-FunctionDecl 0xb7ce0 <c_ex1.c:1:1, line:5:1> func1 'void ()'
| `-CompoundStmt 0xb7f78 <line:1:14, line:5:1>
|   |-DeclStmt 0xb7de8 <line:2:2, col:7>
|   | `-VarDecl 0xb7d90 <col:2, col:6> x 'int'
|   |-DeclStmt 0xb7e88 <line:3:2, col:11>
|   | `-VarDecl 0xb7e10 <col:2, col:10> y 'int'
|   |   `-IntegerLiteral 0xb7e68 <col:10> 'int' 1
|   `-BinaryOperator 0xb7f50 <line:4:2, col:8> 'int' '='
|     |-DeclRefExpr 0xb7ea0 <col:2> 'int' lvalue Var 0xb7d90 'x' 'int'
|     `-BinaryOperator 0xb7f28 <col:6, col:8> 'int' '+'
|       |-ImplicitCastExpr 0xb7f10 <col:6> 'int' <LValueToRValue>
|       | `-DeclRefExpr 0xb7ec8 <col:6> 'int' lvalue Var 0xb7e10 'y' 'int'
|       `-IntegerLiteral 0xb7ef0 <col:8> 'int' 2
`-FunctionDecl 0xb7fd0 <line:7:1, line:9:1> func2 'void ()'
  `-CompoundStmt 0xff0b8 <line:7:14, line:9:1>
    `-CallExpr 0xff090 <line:8:2, col:8> 'void'
      `-ImplicitCastExpr 0xff078 <col:2> 'void (*)()' <
          FunctionToPointerDecay>
        `-DeclRefExpr 0xb8070 <col:2> 'void ()' Function 0xb7ce0 'func1' '
            void ()'
```

Listing 3.2: Simplified AST of example given in listing 3.1

For every basic node type (i.e. `Decl`, `Stmt`, `Expr`, `Type`), one such a *Converter* or *Visitor* is implemented: DeclVisitor, StmtConverter, ExprConverter, and the TypeConverter. The Converters use the Visitor-pattern to traverse the Clang AST. The Converters implement visitor methods to convert the individual nodes of the Clang AST. These visitor methods need to know how to traverse the specific Clang AST node, i.e. which child nodes are available and how to access them.

During the traversal of the Clang AST the active converter might change as the visitor methods passes the conversion of a child node to responsible converter. This means when during the conversion of a `Stmt` node the StmtConverter encounters a child which is a `Expr` node, the conversion of the child node - the `Expr` node - is handled by the ExprConverter.

In the following Section 3.3.2 we cover in detail how declarations are converted. The three main declarations we need to convert are:

- `TypeDecl` - node for declarations of user-defined record types like `struct`, `class`, or `union` and for `typedefs`
- `FunctionDecl` - node for declarations of functions, and as well for C++ member functions, constructors or destructors as well
- `VarDecl` - node for declarations of variables like parameters, global variables, and local variables

The main entry point into the Clang AST is a `TranslationUnitDecl`. First we convert all type declarations (`TypeDecl`), followed by the variable declarations for global variables (`VarDecl`) and finally function declarations (`FunctionDecl`). Variable declarations for parameters and local variables (`VarDecl`) are converted ad-hoc during the traversal of the declaration statement they are declared in.

Furthermore we present the Converters for the basic node types i.e. for `Type` nodes - the TypeConverter in Section 3.3.3, for `Stmt` nodes - the StmtConverter in Section 3.3.4 and for `Expr` nodes - the ExprConverter in Section 3.3.5.

As the Clang AST has a rather large number of nodes, we showcase only a selection of the nodes for each basic node type.

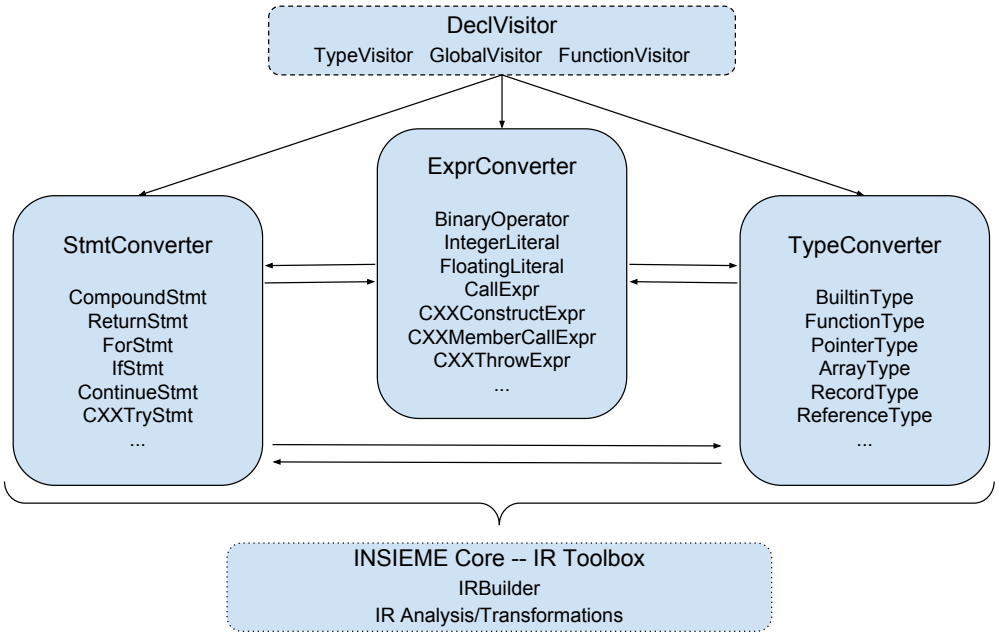

Figure 3.4: Interaction between the main frontend components

In Figure 3.4 we give an overview of the main Frontend components - the different declaration visitors (TypeVisitor, FunctionVisitor, GlobalVisitor), Converters (TypeConverter, StmtConverter, ExprConverter) and the Insieme Core tools - interact.

The declarations encountered during the traversal of a `TranslationUnitDecl` are visited by the declaration visitor (`DeclVisitor`). The child nodes of such a declaration are then converted by the responsible Converter. For example a function declaration has its body as a compound statement. Thus the conversion of the compound statement is handed over to the StmtConverter. The

same is true for expressions found in the body, they are handed over to the ExprConverter, and types are converted by the TypeConverter.

The IR for the various nodes is generated with the help of the Insieme tool box – the Insieme Core. The IRBuilder provides means to build up the various IR statements, expressions and types. Besides that the Insieme Core provides methods to analyse and transform IR fragments.

3.3.2 Converting `Decl` nodes

The conversion of `Decl` nodes starts with the traversal of the Clang AST at the main entrypoint: a `TranslationUnitDecl`. We use a visitor for declarations and declaration contexts. This visitor traverses to all child nodes which are a `Decl` node. If a declaration is a declaration context (e.g. `FunctionDecl`, `NamespaceDecl`, `RecordDecl`, `TranslationUnitDecl`) we recursively visit all contained declarations.

Depending on the child nodes we are interested in, we implement certain visitor methods. For example the TypeVisitor implements only the visitor method for `TypeDecls`, and the FunctionVisitor only the visitor method for `FunctionDecls`.

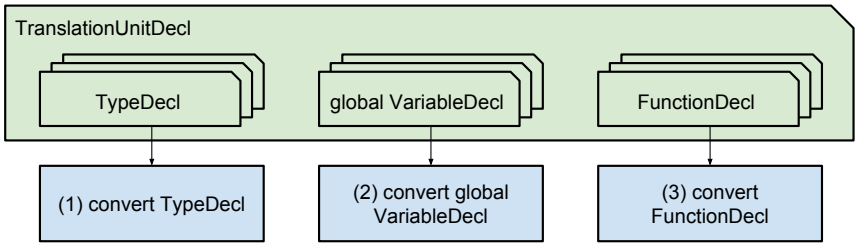

Figure 3.5: Traversing a TranslationUnitDecl

As pictured in Figure 3.5 we start our conversion process by visiting first all type declarations (`TypeDecl`) (1) (e.g. `TypedefDecl`, `RecordDecl`, `EnumDecl`). Thus the visitor methods for these declarations are implemented. In that way we have converted all user defined types in beforehand and can use them in the subsequent conversion process. After converting all user-defined types we continue with all variable declarations (`VarDecl`) of global variables (2). Finally we convert function declarations (`FunctionDecl`) (3). This includes C++ member methods - `CXXMethodDecl`, constructors - `CXXConstructorDecl` and destructors - `CXXDestructorDecl`.

The visitor methods handling these kinds of declarations forward the actual conversion to separate conversion methods (`convertTypeDecl`, `convertFunctionDecl`, `convertGlobalVariable`). These conversion methods expect a declaration node

and return the corresponding IR representation. To alter the conversion the plugin system can be utilized.

Symbolic IR For name-dependent declarations - such as user-defined types, functions, global variables - we generate an IR *symbol* to be used throughout the conversion process. The actual implementation of the declaration is stored along with the symbol in the IRTranslationUnit.

For user defined types we use an IR *generic type* as symbol. The builtin types like char, integer, floating point types are represented by the primitive IR types and need no symbol.

For functions and global variables an IR *literal* with the correct IR type is generated and used as a symbol. For functions we use the function name and the function type for the symbol.

For global variables we extend the variables name by the input source file and line position of the variable declaration in order to create a unique name and avoid name collisions. This unique name, and the type of the global variable is used for the symbol.

For all other variables we do not use a symbol but generate an IR *variable* with the type of the variable declaration. The name of the variable is automatically created. For convenience a *NameAnnotation* can be attached to provide the variable name to the Backend.

Whenever the Clang AST is referring to a declaration, we use the IR symbol instead of the actual implementation. For example see the details for the conversion of a `CallExpr` node for the use of function symbols, or the `TagType` node for the use of type symbols.

The resolution and resolving of the symbolic IR into the logical IR is shown in Section 3.4.

Converting `TypeDecl`

The Clang AST has several nodes inheriting from `TypeDecl`. A major role plays the `TagDecl` - modelling enums, classes, structs and unions - and the `TypedefDecl` - modelling a typedef:

`TagDecl` - is a super class of the `EnumDecl` - for enum types - and `RecordDecl` - for struct/class/union types. The actual conversion into an IR type is handled the by the TypeConverter.

`TypedefDecl` - Typedefs are seen as syntactic sugar giving a different name to a type. Thus we use the actual type. In case of an anonymous type (i.e. enum, struct, class or union) we use the name from the typedef in

the IR as symbol. The actual underlying `Type` node is converted by the TypeConverter.

Such a `TypeDecl` is either named or anonymous. For an anonymous `TypeDecl` we derive a name. That name is used to create a symbol - an IR *generic type* - which is used throughout the conversion process. This symbol is stored in an IRTranslationUnit along with the definition of the type. During the conversion process we only use the symbol. The resulting symbol is stored in a type cache to avoid multiple conversions of the same type.

Converting `VarDecl`

The conversion for variable declarations differs depending on the kind of the variable. We distinguish between global variables and local variables.

- Global variables are converted into an IR *literal* (i.e. the symbol), and an IR *expression* representing the initialization. The resulting IR *literal* is stored in a cache to avoid multiple conversions. In the IRTranslationUnit we store the IR *literal* for the global variable along with the initialization expression. During the conversion process we only use the symbol. After the conversion process is done, and all IRTranslationUnits are merged into a single IRTranslationUnit, all the globals are prepended to the body of the main entry point of the IRProgram (e.g. the `main()` function).
- Local variables are converted into an IR *variable* with the variables type converted by the TypeConverter into an IR type.
- Static (local) variables are modeled in a special IR construct to represent the first initialization at the first execution of the function they are declared in.

Only global variables are converted before the function declarations. Local and static local variables are converted ad-hoc when they are encountered during the conversion of the function declaration they are declared in. To avoid multiple conversions of the same variable, we store the IR result of a variable (i.e. an IR *literal* or IR *variable*) in a cache.

Mutable variables As in C/C++ variables are always mutable, we need to represent this as well in IR. We use the IR extension type *ref*. This represents the reference to an mutable memory location as introduced in Section 2.3. Hence a mutable C/C++ variable is represented as an IR variable with a *ref* type. For example a variable of type `int` has the IR type $ref\langle int\langle 4\rangle\rangle$.

Converting `FunctionDecl`

To convert a function declaration we need to convert the function's type, its parameters and the function body itself. These steps are pictured in Figure 3.6. In IR we express a function declaration as a *lambda expression*. To refer to a function, for example through a function call, we use during the conversion only a symbol - an IR *literal*. Such a symbol is basically the name of the function and its function type. After converting a function declaration we store the symbol along with the implementation in the IRTranslationUnit. To avoid multiple conversions of the same function declaration we use a cache to store the used symbols.

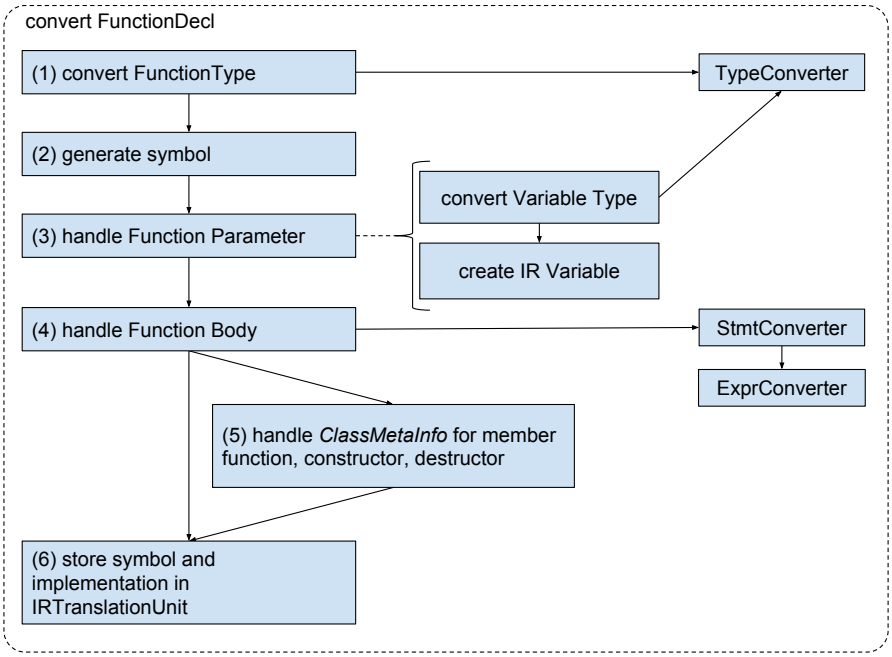

Figure 3.6: Converting a `FunctionDecl`

The conversion process for function declarations starts by converting the function type (1). The particular types for the return type and the parameters are converted into IR types by the TypeConverter. In the case of a C++ member functions we additionally need to extend the parameter list with the parameter for the `this`-pointer, and what kind of member function they are e.g. a constructor, a destructor or an ordinary member function. All this results in an IR *function type*.

For every function we generate a symbol - an IR *literal* - out of its name and type (2). This symbol is stored in the IRTranslationUnit along with the

functions definition. During the conversion process we use only the symbol whenever that function is referenced (e.g. through a function call).

Next we convert the function parameters into IR *variables* (3). These variables are only available in the functions scope.

Following, we continue with the conversion of the functions body (4). As a function body is always a Clang `CompoundStmt` node, this conversion is handed over to the StmtConverter.

To ensure that we are able to reinstanciate all member functions, constructors, and destructor for a certain C++ class, we use the *ClassMetaInfo* annotation (5). Such a *ClassMetaInfo* stores to a class type all the member functions, constructors and destructor. Along with the used IR symbol we store the information if a member functions was declared as `virtual` or `const`. The *ClassMetaInfo* annotation is stored in the IRTranslationUnit, it gets attached to the class type after resolving the symbol with the actual IR implementation in the final resolution step.

Finally we store the symbol for the function declaration along with the functions IR implementation in the IRTranslationUnit (6).

Special cases Some special cases and details are not pictured in Figure 3.6:

- Pure virtual Member function - for pure virtual (or abstract) member functions we generate only a symbol - an IR *literal*. The correct implementation is provided by a super class of the member functions class type.

- `this`-parameter - C++ member functions, constructors and destructor expect an implicit `this`-parameter - representing the object the member function is acting on. Typically this is done by an additional parameter - in the first position - of the member function type. In the converted function body we use an IR *literal* to represent the `this`-variable, when the `this`-parameter (an IR *variable* of the class type) is added to the function type we replace the `this`-literal with the IR *variable* used as a parameter.

- Extern functions - Extern functions only provide the functions prototype and do not have a body. Due to this we can infer that some other translation unit or library provides the implementation of this function. Thus we only generate an IR symbol with the function's name and the function's type. Additionally we annotate the symbol with the information from which header file the declaration came. This *IncludeAnnotation* is needed for the Backend to correctly include the header files for third-party libraries. For more details on third-party libraries see Chapter 4.

- Templated functions - Templated functions and member functions are handled in the Clang AST by providing fully instantiated function (member function) declarations. The handling of the IR symbol differs as the template arguments are used in the name. Thus we do not need any special treatment besides the name handling in order to convert them into IR.

  Listing 3.3 shows a templated function and Listing 3.4 the (simplified - some details are omitted, and the names might differ) resulting IR function. Notice the names, *fun_int*, and *fun_float*, of the generated IR function, representing the template argument.

```
template<T>
void fun(T par) { ... }
...
fun<int>(1);
fun<float>(1.0);
...
```

Listing 3.3: Templated C++ function

```
...
let = fun_int (int<4> par) -> unit { ... }
let = fun_float (real<4> par) -> unit { ... }
...
fun_int(1);
fun_float(1.0);
...
```

Listing 3.4: IR struct for templated C++ function

3.3.3 Converting `Type` nodes

The conversion of `Type` nodes is implemented by the TypeConverter. It uses a visitor pattern implemented by the TypeVisitor provided by the Clang project. We implemented the actual conversion of the different nodes in the visitor methods. We will showcase only a selection of the type nodes the Clang AST provides and we convert. The TypeConverter offers means to alter the conversion through the plugin system. The `Type` nodes are converted into IR *types*. For user-defined types (e.g. struct, class, union) symbols are used during the conversion process, the actual implementation of these types is stored in the IRTranslationUnit.

BuiltinTypes

The builtin types are converted into their corresponding primitive IR type:

- Character types - the basic `char` is represented in IR as a *char*. For wide character support the IR type *wchar* is used with an additional type parameter specifying the width (e.g. 16 or 32 bits).

- Integer types - signed integers (e.g. `short`, `int`, `long`, `__int128_t`) are represented as *int*. For the unsigned types (e.g. `unsigned short`, `unsigned int`, `unsigned long`, `__uint128_t`) *uint* is used. An additional type parameter is used to specify the width in bytes. For example a `short` with 2 bytes - $int\langle 2\rangle$, and a `unsigned int` with 4 bytes - $uint\langle 4\rangle$.

- Floating point types - e.g `float`, `double`, `long double` are represented in IR as *real* type with an additional type parameter specifying the width, for `float` - $real\langle 4\rangle$, `double` - $real\langle 8\rangle$, `long double` - $real\langle 16\rangle$.

- Void type - represented in IR as *unit*.

- Boolean type - represented in IR as *bool*.

FunctionType

A `FunctionType` is converted into an IR *function type*. The return type, and the parameter types are converted separately, and then put together into an IR *function type*. In the special case of having only one parameter of type `void` the parameter type is omitted in the resulting IR *function type*. For variadic arguments, an IR abstract type is used (*VarList*).

For example a function expecting an integer and a float, without a return value: in C/C++ `void fun(int, float)`) and in IR *(*$int\langle 4\rangle$*,* $real\langle 4\rangle$*) -> unit*.

The Clang AST makes no distinction between a function type for a C function or for a C++ member function, constructor or destructor, but INSPIRE does. INSPIRE uses an additional parameter for the IR *function type* to distinguish between constructor, member functions, destructors, or plain functions.

ArrayType

We differ between arrays with a fixed size and arrays which are dynamically sized.

Fixed-size array Array types with a fixed size - i.e. the size is known at compile-time - are represented as a `ConstantArrayType` node in the Clang AST. These nodes are converted into an IR *vector* type, with the element type and the fixed size as type parameters (e.g. *vector⟨elementType, size⟩*).

For a mutable variable of a fixed-sized array we use *ref⟨vector⟨elementType, size⟩⟩*, which represents a mutable IR vector with elements of *elementType*.

Dynamically-sized array For dynamically-sized array types - i.e. the size is not known at compile-time - the Clang AST uses three different nodes (`DependentSizeArrayType`, `IncompleteSizeArrayType`, `VariableArrayType`). These nodes are converted into an IR *array* type with the element type and the dimension of the array as type parameters (e.g. *array⟨elementType, dimension⟩*).

For a mutable variable of a dynamically-sized array we use *ref⟨array⟨elementType, dimension⟩⟩*, which represents a mutable IR array with elements of *elementType*.

PointerType

A pointer type represents a reference to an entity of the referred type. For example an integer pointer type (i.e. `int*`) refers to a memory location representing an integer value (i.e. an `int`).

As a pointer type is a reference to a memory location, we use the IR *ref* type to model C/C++ pointers types.

The referred type is converted on its own by the TypeConverter and is subsequently used to build the correct IR representation of the given pointer type.

There are two options to model pointer types in IR, depending on if the pointer is pointing to a scalar or an array:

Pointer-to-scalar A memory location of a scalar would be represented as *ref⟨type⟩*. Thus a pointer type pointing to a scalar could be modeled as *ref⟨ref⟨type⟩⟩* as lvalue-type and *ref⟨type⟩* as rvalue-type.

Pointer-to-array A memory location of an array would be represented as *ref⟨array⟨type,1⟩⟩*. Thus a pointer type pointing to an array could be modeled as *ref⟨ref⟨array⟨type,1⟩⟩⟩* as lvalue-type and *ref⟨array⟨type,1⟩⟩* as rvalue-type.

But just from a C/C++ pointer type (e.g. `int*`) we can not determine if the memory location the pointer type is referring to is a scalar (i.e. a `int`) or an array (i.e. `int[]`). As we can model a scalar as an array with only one element, the pointer-to-array is the more general representation. This results

in all pointer types to be represented as *ref⟨array⟨type,1⟩⟩* (rvalue). A pointer variable is then represented in IR as *ref⟨ref⟨array⟨type,1⟩⟩⟩* (lvalue).

Special cases In C/C++ we have two special cases of pointer types, the pointer to a void type, and the pointer to a function type.

Void pointer – A void pointer type (i.e. `void*`) represents a reference to a memory location containing a value of any (or unknown) type. In IR we use the *any* type to represent any type. Thus the void pointer is represented as *ref⟨any⟩* (rvalue). Thus a variable of void pointer type is represented as *ref⟨ref⟨any⟩⟩* (lvalue).

To represent an array of void pointers we either use the IR *vector* or *array* type. Depending on the array if it is dynamically-sized or fixed-sized:

- A dynamically-sized array of void pointers is represented as *ref⟨ref⟨array⟨ref⟨any⟩,1⟩⟩⟩*.
- A fixed-sized array of void pointers is represented as *ref⟨vector⟨ref⟨any⟩,size⟩⟩*.

Function pointers – A function pointer type represents a reference to an actual function. In Listing 3.5 a function pointer to a function expecting an integer, and having no return value is shown.

```
1 void (*fp)(int);
```

Listing 3.5: C/C++ function pointer type example

Thus in IR we simply represent it as an IR *function type* (rvalue). A variable of such a function pointer is then represented as *ref⟨function type⟩* (lvalue).

The same applies for dynamically-sized and fixed-sized arrays of function pointers:

- A dynamically-sized array of function pointers is represented as an (mutable) array of functions. These are converted into an IR array of function types: *ref⟨array⟨function type,1⟩⟩*. A variable of a dynamically-sized array of function pointers is then represented as *ref⟨ref⟨array⟨function type,1⟩⟩⟩*.
- A fixed-sized array of function pointers is represented as a vector of functions: *vector⟨function type,#elem⟩*. A variable of a fixed-sized array of function pointers is then represented as *ref⟨vector⟨function type,#elem⟩⟩*.

Pointer type	lvalue	rvalue
Pointer-to-scalar	*ref⟨ref⟨array⟨type,1⟩⟩⟩*	*ref⟨array⟨type,1⟩⟩*
Pointer-to-array	*ref⟨ref⟨array⟨type,1⟩⟩⟩*	*ref⟨array⟨type,1⟩⟩*
Void pointer (`void*`)	*ref⟨any⟩*	*ref⟨ref⟨any⟩⟩*
Function pointer	*function type*	*ref⟨function type⟩*

Table 3.1: IR representations of pointer types

VectorType

The `VectorType` node represents a generic vector type used for SIMD operations. In IR we represent this as a (abstract) type: *simd⟨vector⟨elementType,size⟩⟩*. Additionally to the type, we provide unary and binary operators to be used with the IR type. For the conversion of the operators see the sections on `UnaryOperator` 3.3.5 and `BinaryOperator` 3.3.5.

TagType

The `TagType` represents `struct`, `class`, `union`, `enum` types. They are organised in subclasses, for `struct`, `class`, `union` a `RecordType` is used. For enumerations (i.e. `enum`) an `EnumType` is used.

`RecordType` For `struct/class` types we create an IR *struct* type, consisting of all the fields and their types (e.g. *struct NAME* { *int⟨4⟩ field1; real⟨4⟩ field2;* }). The same holds for `union` types, for these we generate an IR *union* type, with all the member fields (e.g. *union NAME* { *int⟨4⟩ field1; float⟨4⟩ field2;*}).

For C++ `struct/class` types, besides the member fields also the inheritance relations to the types parents are added to the IR type.

We also generate an IR symbol for `struct/class/union` types - an IR *generic* type. That symbol is used during the conversion process instead of the actual type. The type implementation is stored along with its symbol in the IRTranslationUnit. This symbol also eases the resolving of recursive types. The final resolution of recursive structs is done when all IRTranslationUnits were merged together. See Section 3.4 for details on the resolution process. Typically a `RecordType` has a name, which is used for the IR symbol and the IR type definition.

Recursive `RecordType` Example In Listing 3.6 we give a simple example of a recursive `struct` type (e.g. used for a linked list) in C. From that `struct`

type the *StructType* symbol is generated. This symbol is also used in the implementation to model the pointer to another `StructType` object. This pointer introduces the recursion in the type. In Listing 3.7 the corresponding entry in the IRTranslationUnit is shown.

After the resolution of the symbols the correct IR *recursive* type is used for such cases.

```
struct StructType {
    int data;
    StructType* next;
};
```

Listing 3.6: Simple recursive C/C++ `struct` type

```
StructType : struct StructType {
    int<4> data;
    ref<array<StructType,1>> next;
}
```

Listing 3.7: IRTranslationUnit entry of a simple recursive *struct* type

`EnumType` For enumeration (i.e. `enum`) types we generate an abstract IR type which contains a unique name of the enum and all the enums constants with their initial value. Enum constants are represented with an abstract IR type of their own, consisting of the constants name and the constants value.

In Listing 3.8 the abstract IR type used to represent enum constants is shown. *constantName* is a unique name of the constant, and *constantValue* is an integer value representing the enum constants value.

```
__insieme_enum_constant<constantName, constantValue>
```

Listing 3.8: IR construct for enum constant

In Listing 3.9 the abstract IR type used to represent enum types is shown. *enumName* is the enum types name. The shown enum type consists of two constants: *someConstant1*, *someConstant2*

Templated `RecordType` The Clang AST handles templated record types similar to the way templated functions and member functions are treated, as described in 3.3.2. The Clang AST provides already the fully instantiated type declarations for templated record types.

```
__insieme_enum<
  enumName,
  __insieme_enum_constant<someConstant1, initValue1>,
  __insieme_enum_constant<someConstant2, initValue2>
>
```

Listing 3.9: IR construct for enum type

Listing 3.10 shows a templated C++ struct and Listing 3.11 the (simplified - some details are omitted, and the names might differ) resulting IR. Notice the member field *elem* which is already typed to an IR *integer*

```
template<T>
struct TemplatedStruct {
  T elem;
  ...
};
...
TemplatedStruct<int> obj;
...
```

Listing 3.10: Simple templated C++ struct

```
...
struct TemplatedStruct_int < elem:int<4> > obj;
...
```

Listing 3.11: IR struct for simple templated C++ struct

ReferenceType

`ReferenceType` is a C++ specific type node, modelling the reference type (e.g. `int&`). This is converted into a special IR construct for references (Listing 3.12) and constant references (Listing 3.13).

```
struct { ref<'a> _cpp_ref; };
```

Listing 3.12: IR construct for C++ reference type

Additionally to these two constructs, operators are provided to manipulate and access these constructs.

```
struct { src<'a> _const_cpp_ref; };
```

Listing 3.13: IR construct for C++ const reference type

This special IR construct is needed to keep compatibility with third-party libraries, as the reference type could be represented with the *ref*-type as well, similar to the pointer type. But this would result in the same code generated by the Backend - most likely as a pointer- and if the third-party library expects a reference the generated code would be incompatible.

3.3.4 Converting `Stmt` nodes

The conversion of `Stmt` nodes is implemented by the StmtConverter. As INSPIRE provides similar constructs to represent statements as the nodes used by the Clang AST, the conversion of statement nodes is rather straight forward. The StmtConverter takes Clang AST `Stmt` nodes and converts them into IR *statements*.

CompoundStmt

A Clang AST `CompoundStmt` is turned into an IR *CompoundStmt*. All the contained statements are converted by the StmtConverter and put in the resulting IR *CompoundStmt*.

IfStmt

A Clang AST `IfStmt` is turned into an IR *IfStmt*. The condition expression, the `then`-branch and `else`-branch are converted separately and then put together into the IR *IfStmt*. A cast is added for condition expressions which are not of type *bool*.

In Listing 3.14 we give an example for a simple while-loop in C/C++, and in Listing 3.15 we show the corresponding while-loop in IR.

ForStmt

A Clang AST `ForStmt` is turned into an IR *ForStmt* but only if it is a true for-loop. This means it needs to be able to be converted into a count-controlled loop, and have no early-exit, i.e. no `break`, or `continue` statement in the loop-body.

The children of a `ForStmt` node are converted by the corresponding converter. The children include besides the loop-body itself, as well the possible initialization/condition and increment expressions.

```
if(condition1) {
  if(condition2) {
    // firstIf then-body
  }
} else {
  if(condition3) {
    // thirdIf then-body
  } else {
    // thirdIf else-body
  }
}
```

Listing 3.14: If-statement in C/C++

```
if(condition1) {
  if(condition2) {
    // firstIf then-body
  };
} else {
  if(condition3) {
    // thirdIf then-body
  }
  else {
    // thirdIf else-body
  };
};
```

Listing 3.15: If-statement from Listing 3.14 in IR

If the for-loop can not be turned into a count-controlled loop, it is rewritten as a condition-controlled while-loop.

In Listing 3.16 we give an example for an for-loop which will be re-written into an while-loop in IR. In Listing 3.17 we show the resulting while-loop in IR.

```
for(int i=0;i<10;i++) {
  if(i%2 == 0) {
    i++;
  }
}
```

Listing 3.16: C/C++ For-Loop which gets rewritten into While-Loop

```
{
  ref<int<4>> i = < init code >;
  i := 0;
  while(*i<10) {
    if(i % 2 == 0) {
      i++;
    }
    i++;
  };
};
```

Listing 3.17: Rewritten C/C++ While-loop from Listing 3.16 (in IR)

WhileStmt

A Clang AST `WhileStmt` is turned into an IR *WhileStmt*. The condition expression is converted by the ExprConverter, and the loop-body is converted by the StmtConverter and represented as an IR *CompoundStmt*. For condition expressions which are not of type *bool* a cast is added.

In Listing 3.18 we give an example for a simple while-loop in C/C++, and in Listing 3.19 we show the corresponding while-loop in IR.

```
while(condition) {
  // while-body
}
```

Listing 3.18: C/C++ While-Loop

```
while(condition) {
  // while-body
};
```

Listing 3.19: While-loop from Listing 3.18 in IR

Switch/Case

The Clang AST uses different nodes to model the `switch` - `SwitchStmt`- and the `case` - `SwitchCase`.

The IR switch statement does not provide a fall-through if there is no `break`-statement between different cases. To model this semantics in IR the statements of consecutive cases are copied into their own IR cases. In IR a `default` case is

always added, if there is no `default`-case in the input code, it is just an empty IR *CompoundStmt*.

For example Listing 3.20 shows a switch/case with fall through, which turns into the IR representation as given in Listing 3.21

```
switch(condition) {
  case1:
  case2:  stmtOfCase2;
      break;
}
```

Listing 3.20: Switch/Case with fall-through in C

```
switch(condition) {
  case1: stmtOfCase2; break;
  case2: stmtOfCase2; break;
  default: {}
}
```

Listing 3.21: Switch/Case with fall-through in IR

ContinueStmt

A Clang AST `ContinueStmt` is turned into an IR *ContinueStmt*.

BreakStmt

A Clang AST `BreakStmt` is turned into an IR *BreakStmt*.

ReturnStmt

A Clang AST `ReturnStmt` is turned into an IR *ReturnStmt*. The IR *ReturnStmt* has the type of the return expr. If the `return` has no return expression, the IR *ReturnStmt* is of type *unit* and returns a *unit literal*.

Exceptionhandling – try/catch

C/C++ exception handling is represented by the Clang AST with the `CXXTryStmt` and the `CXXCatchStmt` nodes. These nodes are turned into IR *TryStmt* and a *CatchClause*. The parameters of the `catch` statements are represented as `VarDecls` and are converted similar to the parameters of a

function declaration. The ellipsis (...) indicating a catch-all is represented as a parameter of type *any*.

The `throw` is an expression, thus it is explained in the following section when the ExprConverter is detailed.

For an example see Listing 3.22. It shows a try-catch with a catch-all clause, which turns into the IR representation as given in Listing 3.23.

```
try {
  // try-body
} catch(...) {
  // catch-all-body
}
```

Listing 3.22: Try-Catch with a catch-all caluse in C++

```
try {
  // try-body
} catch(any v1){
  // catch-all-body
}
```

Listing 3.23: Try-Catch with catch-all clause in IR

3.3.5 Converting `Expr` nodes

The conversion of `Expr` nodes is implemented by the ExprConverter. It uses the visitor pattern implemented by the `StmtVisitor` (provided by the Clang project). The TypeConverter offers means to alter the conversion - plugin system. The `Expr` nodes are converted into IR *expressions*.

Literals

The Clang AST uses several nodes to represent literals of different types, e.g. `IntegerLiteral`, `FloatingLiteral`, `CharacterLiteral`, `StringLiteral`.

- Integer literal - is converted into an IR *literal* of integer type (*int* with the correct width).
- Floating-point literal - is converted into an IR *literal* of floating-point type (*real* with the correct width).

- Character literal - is converted into an IR *literal* of character type (*char* with the correct width). Wide characters are converted into *wchar* with their corresponding width.
- String literal - as a string literal can be seen as a fixed-sized array of characters, string literals are converted into an IR *literal* with type

  vector⟨charType,size⟩. As the string can use different character types we can use for the *charType* a normal character type (i.e. *char*) or a wide character type (i.e. *wchar*).
- Boolean literal - `true` and `false` are converted into IR *literals* of IR type *bool*.

Casts

The Clang AST has several nodes to model casts. There are two categories the implicit casts and the explicit cast. The explicit casts themselves are then separated into different nodes to differ between an explicit C cast and explicit C++ casts like `static_cast`, and `dynamic_cast`.

Additionally to these nodes the Clang AST uses an enumeration to differentiate between the different kinds of cast. As there are over 50 different kinds of casts used by the Clang AST we demonstrate only a few.

- Casts between Integrals, Floating point and boolean are realised with separate IR operators. For example a cast from an unsigned integer to a signed integer the *uint.to.int* operator is used. This IR operator expects a source type and a target precision. In the case for *uint.to.int* the prototype is *(uint⟨#a⟩,#b) -> int⟨#b⟩*. Where #a and #b are integer type parameters specifying the width of the type.

  When the cast is between the same type but the precision changes, an IR operator to adjust the precision is used. For examples from `short` to `int` integers the *int.precision* operator is used.

  In the case for *int.precision* the prototype is *(int⟨#a⟩,#b) -> int⟨#b⟩*. Where #a and #b are integer type parameters specifying the width of the type.
- Casts between pointer types are called by the Clang AST a `BitCast`. These casts are modeled in general with the IR *ref.reinterpret* operator.
- Casts between integer and pointers are modeled by an IR *int.to.ref* and and *ref.to.int* operator. As there are no negative integer values for pointers this operator expects a *uint*.

C++ specific casts Whenever possible we use the basic IR operators but for certain cases we need to use special operators as there is additional implicit behaviour beside the casting.

- A cast from a sub class to a super class is called by the Clang AST a `DerivedToBase`. This can be modeled with the IR *ref.narrow* operator.
- A C++ dynamic cast - `dynamic_cast<type>` - is modeled with an IR operator of its own: *dynamic_cast*. This is needed as there is implicit behaviour with the dynamic cast as it checks at runtime the types and if the cast is legal, otherwise it throws an exception.
- A cast from a base to derived class type is typically done with a static cast - `static_cast<type>`. This is modeled in IR with an operator of its own: *static_cast*.

MemberExpr

The Clang AST `MemberExpr` node models an access to a member of a record type (like `struct`, `union`, or `class`). For this node we need to convert the base object and construct the access to the member. Depending on the access type either by-arrow (i.e. `base->member`) or by-dot (i.e. `base.member`) we use different IR operators as the type of the base object differs.

Access-by-arrow In this case the base objects type is a pointer-type and, thus we use the IR *composite.ref.elem* operator which is implemented with the *ref.narrow* operator.

Access-by-dot In this case the base objects type is a value-type and use the IR *composite.member.access* operator.

Binary Operators

C/C++ provides several different builtin binary operators:

- comparison operators (`<`, `>`, `<=`, `>=`, `==`, `!=`, `&&`, `||`)
- bitwise and shift operators (`&`, `^`, `|`, `<<`, `>>`) and their compound assignment variant (`&=`, `^=`, `|=`, `<<=`, `>>=`)
- arithmetic operators (`+`, `-`, `*`, `/`, `%`) compound assignment variant (`+=`, `-=`, `*=`, `/=`, `%=`)
- assignment operator (`=`)

For all this operators the IR provides an equivalent operator. The compound (assignment) operators, i.e. operators of the form `lhs op= rhs;`, are rewritten into `lhs = lhs op rhs;` in IR.

The generated IR not only depends of the operator but also depends on types of the operands (`lhs`/`rhs`).

In general we convert first the left-hand-side and right-hand-side expressions and then we convert the correct operator and generate a call-expression to the IR operator with the left-hand-side and right-hand-side expressions as arguments.

Assignment In IR to model mutable variables we use the IR *ref* type. An assignment of a value to a variable can only happen for variables of *ref* type. For these assignments the IR *ref.assign* operator is used.

Pointer arithmetic The IR operator *arrayView* is used to model pointer arithmetics. For pointer artihmetics we need one of the operands to be a pointer and the operator needs to be an additive operator (`+, -`). A special case of pointer arithmetics is the pointer distance. This occurs with a subtraction operator, and both operands being pointers, i.e. `int x = ptr1-ptr2;` . This is modeled with the IR *arrayRefDistance* operator.

Logical operators As C/C++ uses short-circuit evaluation for logical-and (i.e. `&&`) and logical-or (i.e. `||`), the IR operators need to represent this semantics. This is implemented by lazy evaluating the right-hand-side operand.

Comparison operators For every primitive IR type, IR provides the corresponding comparison operators.

Shift operators For every primitive IR type, IR provides the shift operators.

SIMD vector operators For vector types, we use distinct IR operators to model the possibility of running them on SIMD hardware (Single Instruction, Multiple Data). We have SIMD operators for the binary operators (`+, -, *, %, /, &, |, ^, <<, >>`), comparison operators (`==, !=, <, >, <=, >=`) and unary operators (`~, -`).

Unary Operators

For the C/C++ unary operators the IR provides an equivalent operator or we use a IR construct to model the behaviour.

Increment and decrement operators C/C++ provides a *post*-increment and a *post*-decrement as well as a *pre*-increment and a *pre*-decrement operator. To model the specific behaviour of these operators, that is for the post-increment/decrement returning the value of the expression before the increment/decrement occured and for the pre-increment/decrement returning the value after the increment/decrement occured, we use the following constructs:

- Post-increment/decrement are modeled in IR explicitly as:

C/C++ operator	IR construct
`a++`	`tmp=a; a = a+1; tmp;`
`a--`	`tmp=a; a = a-1; tmp;`

- Pre-increment/decrement are modeled in IR explicitly as:

C/C++ operator	IR construct
`++a`	`a = a+1; a;`
`--a`	`a = a-1; a;`

A special case is when we have post/pre increment or decrement on pointer types. In IR we represent this with the IR *arrayView* operator.

Other unary operators

- Bitwise not (`~`) - represented as an IR *not* operator. For vector types we use a distinct SIMD operator.
- Logical not (`!`) - represented as an IR *lnot* operator, for non-boolean types we need to cast the sub-expression, representing the operand, to the IR *bool* type.
- Unary minus (`-`) - for vector types we use a distinct SIMD operator, for expressions of all other types we get the sign inverted (basically subtracting the expression from 0: `-expr => 0 - expr`)
- Dereference operator (`*`) - for IR *array/vector* types an array access of element 0 is modeled. For the *ref* type we use the IR *deref* operator
- Address-of (`&`) - is represented by the IR *scalar.to.array* operator. As the address-of operator returns the address as a pointer and pointer values are represented in IR as *ref⟨array⟨type,1⟩⟩*. Thus the IR *scalar.to.array* operator turns a scalar (*ref⟨type⟩*) into an array/pointer (*ref⟨array⟨type,1⟩⟩*).

DeclRefExpr

Whenever the Clang AST refers to another declaration a `DeclRefExpr` node is used. For example a function call is modeled as a `CallExpr` node with a `DeclRefExpr` pointing to the actual function declaration of the callee.

The same principle applies for variables, enum types or in the case of C++ for fields of structs and classes.

As we are using only IR symbols during the conversion we need to look up the correct symbol for a given declaration or convert the given declaration into an IR symbol:

- Function declarations are converted into an IR symbol, which is basically an IR *literal*
- Global variable declarations for global variables are converted into an IR *literal*
- Local variable declarations for local variables are converted into an IR *variable*

CallExpr

The `CallExpr` represents all calls to normal functions. C++ member functions are handled in a separate `CXXMemberCallExpr` node.

We need to convert the callee, the arguments and generate an IR *call expression*. The conversion of the callee is handled by the ExprConverter. Typically the callee is a `DeclRefExpr` refering to a function declaration. This returns a *literal* - the IR symbol - representing the callee. The arguments are as well converted by the ExprConverter.

Listings 3.24 and 3.25 give a short example how a C function call is converted into INSPIRE.

```
1 void func1(int arg1, float arg2, char arg3) {
2   //function body
3 }
4
5 ...
6 func1(1, 2.0, 'c'); //call to func1
7 ...
```

Listing 3.24: C function call

The calls to the function `func1` are converted into an IR *call expression*, calling an IR *literal* , of type $(int\langle 4\rangle, real\langle 4\rangle, char) -> unit$, representing `func1`. The associated implementation is stored in the IRTranslationUnit.

Listing 3.25 shows the IRTranslationUnit entry for the implementation and the IR *call expression*.

```
//implementation stored in IRTranslationUnit
let func1 = (int<4> v1, real<4> v2, char v3) -> unit
{ /* function body */ }
...
func1(1, 2.0, 'c');
...
```

Listing 3.25: Call to `func1` from Listing 3.24 in IR

CXXMemberCallExpr

The `CXXMemberCallExpr` node inherits from the `CallExpr` node. Similar to the `CallExpr` node we need to convert the callee, the arguments and build an IR *call expression*.

Additionally we need to take care of the `this`-object. This means we need to convert it and add it to the arguments. By convention is the `this`-object the first parameter.

Listings 3.26 and 3.27 give a short example how a C++ member function call is converted into INSPIRE.

```
void Class::memberFunc(int arg1, float arg2)
{ //function body }
...
//call to Class::memberFunc
someClass_object.memberFunc(1, 2.0);
...
```

Listing 3.26: C++ member function call

The calls to the member function `memberFunc` are converted into an IR *call expression*, calling an IR *literal*, of type $(ref\langle Class\rangle, int\langle 4\rangle, real\langle 4\rangle, char) -> unit$, representing `memberFunc`. Note that the first parameter of the function type represents the `this`-object. The associated implementation is stored in the IRTranslationUnit. Additionally a member function entry is added to the *ClassMetaInfo* for the `Class` type.

Listing 3.27 shows the IRTranslationUnit entry for the implementation and the IR *call expression*.

```
1 //implementation stored in IRTranslationUnit
2 let memberFun =
3 mfun Class this :: (int<4> v1, real<4> v2) -> unit
4 { /* function body */ }
5 ...
6 //call to Class::memberFunc
7 memberFunc(someClass_object, 1, 2.0);
8 ...
```

Listing 3.27: Call to `Class::memberFunc` from Listing 3.26 in IR

CXXOperatorCallExpr

The `CXXOperatorCallExpr` node inherits from the `CallExpr` node. C++ allows to overload operators, this happens either as member function or as non-member function. Depending on which case we have, the arguments (left-hand-side, right-hand-side) and possibly the this-object need to correctly handled.

After converting the callee and the arguments we generate an IR *CallExpression* calling the converted callee with the converted arguments.

Operator as member function If the overloaded operator is written as member function we need to handle the arguments correctly. This depends on what arity the operator is, unary, binary or the special case of the function call operator (i.e. `operator()`.

Some operators are only allowed to be overloaded as member functions (`=, ->, (), [], ->*, new, new[], delete, delete[]`).

- For unary operators there is only one argument which is then the `this`-argument to the operator member function.
- For binary operators there are two arguments, the first one is seen as the left-hand-side argument, this is the `this`-argument for the operator member function. The second argument is then the right-hand-side argument of the operator.
- For the function call operator there is a special case, as the first argument is seen as the `this`-argument and all the other arguments are the arguments of the function call.

Operator as non-member function If the overloaded operator is written as a non-member function the argument handling changes slightly, as there is no `this`-argument to take care of. Thus the arguments are handled like it would be with a normal function call. For unary operators the first argument is seen as the right-hand-side (for unary prefix operator), and left-hand-side (for unary postfix operator) of the operator. For binary operators the first argument is seen as the left-hand-side and the second argument as the right-hand-side of the operator.

CXXConstructExpr

A `CXXConstructExpr` node models the call to a C++ constructor in the Clang AST.

Basically we do the same as for the member function call, convert the callee, and generate an IR *call expression* with the converted arguments.

One big difference is: there is not yet a `this` object, as it is constructed and initialized with the `CXXConstructExpr`. Thus the `this`-argument is an undefined IR variable of the class type. This represents the memory location on the stack.

Listings 3.28 and 3.29 give a short example how a C++ constructor call is converted into INSPIRE.

```
1 void Class::Class(int arg1, float arg2)
2 { //constructor body }
3 ...
4 //constructor call instantiating an object
5 Class object(1, 2.0);
6 ...
```

Listing 3.28: C++ constructor call

The calls to the constructor for the class `Class` are converted into an IR *call expression*, calling an IR *literal*, of type $(ref\langle Class\rangle, int\langle 4\rangle, real\langle 4\rangle) -> unit$, representing `Class::Class` constructor. Note that the first parameter of the function type represents the `this`-object. The associated implementation is stored in the IRTranslationUnit. Additionally a constructor entry is added to the *ClassMetaInfo* for the `Class` type. As already explained the `this`-object still needs to be created, so the at the call site (i.e. the IR *call expression*) we create a undefined IR *variable* for the `this`-object.

Listing 3.29 shows the IRTranslationUnit entry for the implementation, the IR *call expression* and the undefined IR *variable* as argument.

```
1
2  let Class =
3  ctor Class this :: (int<4> v1, real<4> v2) -> unit
4  { //constructor body }
5  ...
6  //constructor call instantiating an object
7  decl ref<Class> object =
8  Class(
9  //undefined IR variable for this-object
10   var(undef(Class)),
11 //other constructor arguments
12   1, 2.0
13 );
14 ...
```

Listing 3.29: Call to constructor of class `Class` from Listing 3.28 in IR

Non-userprovided constructors For non-userprovided default and copy-constructors we use a dummy implementation which is ignored by the Backend, i.e. the Backend does not generate code for the dummy implementation. As the C++ standard [15] defines that for non-userprovided default and copy-constructors the compiler provides an implementation, hence this is left to the actual backend compiler (e.g. gcc).

CXXNewExpr

The `CXXNewExpr` node models the `new` operator for objects, and the `new[]` operator for object arrays.

With the `new`/`new[]` operators, memory is allocated on the heap. Additionally if the type we allocate memory for, is a class type, the correct constructor is called to initialize the allocated memory. We use the IR *ref.new* operator to model memory allocation on the heap.

Allocating single object - `new` operator The `CXXNewExpr` node has for class types the constructor as a child represented as a `CXXConstructExpr`. The `CXXConstructExpr` is converted on its own. The `this`-variable is then represented by a call to the IR *ref.new* operator representing an allocation on the heap. For non-class types there is no constructor provided. As the `new` operator returns a pointer to the newly allocated memory we need to use the IR *scalar.to.array* operator to turn the scalar/single object into the correct array/-pointer type.

Allocating array object - `new[]` operator For class types we get the constructor expression and generate a derived IR *ArrayCtor* operator. This IR operator takes an allocator-function - in general the IR *ref.new* operator, a constructor, and the size of the array in elements. This IR operator models the allocation of all the elements, and calls the constructor for every element. For non-class type arrays we use the IR operators *ref.new* and *array.create*.

CXXDeleteExpr

With the `CXXDeleteExpr` node the `delete` operator for objects, and the `delete[]` operator for object arrays is represented. With `delete`/`delete[]` heap allocated memory is deallocated, and additionally for class types the destructor is called. We use the IR *ref.delete* operator to model deallocation of heap memory.

For non-class types we use the IR *ref.delete* operator in both cases, the non-array form (`delete`) and the array form (`delete[]`).

Deallocating single object - `delete` For deallocating an object of class type we need a destructor call before deallocating the memory. The destructor call is simply an IR *CallExpr* to the IR *function* representing the class types destructor. Then the deallocation is represented as a call to the IR *ref.delete* operator.

Deallocating array - `delete[]` For deallocating an array of objects of class type we need a destructor call for all elements of the array before deallocating the memory. This is represented by a derived IR *ArrayDtor* operator. This IR operator takes the destructor, deallocator-function - in general the IR *ref.delete* operator, and the expression to be deleted. This IR operator models the destructor call for all elements and the deallocation of all the elements.

CXXThisExpr

The `CXXThisExpr` node represents the `this`-pointer used in member functions.

In IR we use a *literal* (i.e. *this*) with the correct class type. The class-type is converted by the TypeConverter.

When a member function, constructor, or destructor declaration gets converted (as detailed in 3.3.2), we replace the *this-literal* by the actual parameter *variable* used in the member function, constructor, or destructor.

CXXThrowExpr

The `CXXThrowExpr` node represents a `throw` expression used to raise an exception. Typically the `throw` expression is surrounded by a try-catch block.

In IR we use a *ThrowStatement* but ExprConverter needs to return an IR *expression*. Thus we need to wrap the *ThrowStatement* into a *lambda expression* and call the *lambda expression*.

Temporaries and expressions with cleanups in C++

The Clang AST uses several nodes to model temporaries and expressions with cleanups in C++:

- The `CXXTemporary` node is used to identify a C++ temporary.
- The `CXXBindTemporaryExpr` node is used whenever an expression is bound to a temporary (i.e. `CXXTemporary`).
- The `MaterializeTemporaryExpr` node is used whenever a temporary value needs to be materialized - i.e. written into memory.
- The `ExprWithCleanups` node represents an expression which needs to run some cleanup after the expressions evaluation. For example a C++ temporary which needs to run a destructor at the end of its lifetime. This cleanup (e.g. a destructor call) is not explicitly modeled in IR but left as implicit behaviour.

3.3.6 Examples

To exemplify the conversion process we give two examples. At first a rather simple one written in C and a second one written in C++.

C/C++ function example It consists of one translation unit with two functions, `func1` and `func2`. In Listing 3.30 we show the C input code.

From the C code in Listing 3.30 we generate a Clang AST as shown in Listing 3.31. This Clang AST consists of one `TranslationUnitDecl` with two `FunctionDecl` nodes. A `FunctionDecl` (`FunctionDecl` nodes are handled by `convertFunctionDecl` in the DeclarationVisitor).

A `FunctionDecl` has a function body, which is modelled in a `CompoundStmt` (`Stmt` nodes are handled by the StmtConverter). A `CompoundStmt` is composed by other `Stmt` and `Expr` nodes (`Expr` nodes are handled by ExprConverter). In the `Stmt`/`Expr` nodes, `Type` nodes are used (`Type` nodes are handled by the TypeConverter).

```
1 void func1() {
2   int x;
3   int y = 1;
4   x = y+2;
5 }
6
7 void func2() {
8   func1();
9 }
```

Listing 3.30: C function with a Call

```
1  TranslationUnitDecl
2  ... Clang internal declaraions ..
3  |-FunctionDecl func1 'void ()'
4  | `-CompoundStmt
5  |   |-DeclStmt
6  |   | `-VarDecl x 'int'
7  |   |-DeclStmt
8  |   | `-VarDecl y 'int'
9  |   |   `-IntegerLiteral 'int' 1
10 |   `-BinaryOperator 'int' '='
11 |     |-DeclRefExpr 'int' lvalue Var 'x' 'int'
12 |     `-BinaryOperator 'int' '+'
13 |       |-ImplicitCastExpr 'int' <LValueToRValue>
14 |       | `-DeclRefExpr 'int' lvalue Var 'y' 'int'
15 |       `-IntegerLiteral 'int' 2
16 `-FunctionDecl func2 'void ()'
17   `-CompoundStmt
18     `-CallExpr 'void'
19       `-ImplicitCastExpr 'void (*)()' <FunctionToPointerDecay>
20         `-DeclRefExpr 'void ()' Function 'func1' 'void ()'
```

Listing 3.31: Simplified AST of example given in Listing 3.30

The input translation unit with the two C functions is converted into one IRTranslationUnit containing the corresponding IR. This IRTranslationUnit is shown in Listing 3.32.

In Listing 3.32 one sees the IR representation for the two functions `func1` and `func2`. Function `func2` (shown in line 14-16) uses a IR literal (*func1*) to represent a call to `func1`. After resolving this IRTranslationUnit the IR *literal* is replaced with the actual IR implementation of `func1`.

As there are no user-defined types, global variables, global variables initializers, meta-information for C++ methods, or entry points (i.e. `main` function), the IRTranslationUnit consists only of the functions `func1` and `func2`.

```
Types: ,
Globals: ,
Initializer: ,
Entry Points: {},
MetaInfos: {},
Functions:
func1 : (()->unit) => fun() -> unit {
  decl ref<int<4>> v1 = var(undefined(type<int<4>>));
  decl ref<int<4>> v22 = var(1);
  v1 := v22+2;
}
func2 : (()->unit) => fun() -> unit {
  func1();
}
```

Listing 3.32: IRTranslationUnit resulting from the AST given in Listing 3.31

C++ Class Example To show the conversion of a C++ class with member functions we use a simple `Counter` class. The `Counter` class is composed of one data member - `val`, the current value of the counter and several member functions to access and manipulate the current value. With the member functions `inc`, and `dec` the counter value can be increased respectivly decreased. By default `inc` and `dec` change the value by one. The `reset` member function resets the counter value, by default to zero. With `get` the counter value can be retrieved. Listing 3.33 shows the declaration of the `Counter` class.

```
struct Counter {
  int val;
  Counter(int val = 0);
  void inc(int diff = 1);
  void dec(int diff = 1);
  void reset(int val = 0);
  int get() const;
};
```

Listing 3.33: C++ `Counter` class

At the beginning of the conversion process we convert the C++ class type into an IR *type*. The C++ `Counter` class is encoded in INSPIRE into a type similar as shown in Listing 3.34. Let *Counter* denote the resulting IR *type*.

Following the types, all the (member) functions are converted. Exemplary we present the results at the different stages of the conversion process for one of the member functions: `Counter::inc`. The conversion process begins with the

```
struct <
  val:int<4>
>
```

Listing 3.34: IR type of the `Counter` class

the input code in C++ (Listing 3.35), from which a Clang AST is generated (Listing 3.36). From the Clang AST finally the IR is produced (Listing 3.37).

The implementation in C++ is shown in Listing 3.35. It takes one integer argument and adds it to the `Counter` class data member `val`. The corresponding Clang AST is shown in Listing 3.36. It consists of an `CXXMethodDecl` with the body in an `CompoundStmt` and the parameter as a `ParmVarDecl`. As the parameter has a default value (in the declaration - see Listing 3.33), the `ParmVarDecl` node has a child node to represent this default value - a `IntegerLiteral` node.

```
void Counter::inc(int diff) { this->val += diff; }
```

Listing 3.35: `Counter::inc` member function of the `Counter` class

```
1 CXXMethodDecl inc 'void (int)'
2 |-ParmVarDecl diff 'int'
3 | `-IntegerLiteral 'int' 1
4 `-CompoundStmt <col:25, col:46>
5   `-CompoundAssignOperator 'int' lvalue '+=' ComputeLHSTy='
       int' ComputeResultTy='int'
6     |-MemberExpr 'int' lvalue ->val
7     | `-CXXThisExpr 'struct Counter *' this
8     `-ImplicitCastExpr 'int' <LValueToRValue>
9       `-DeclRefExpr 'int' lvalue ParmVar 'diff' 'int'
```

Listing 3.36: Clang AST for member function `Counter::inc`

The `MemberExpr` (line 6 and 7) models the access of the `val` data member of the `Counter` class. The child node models thereby the object on which the member access is happening. In this case the object is the `this`-object, represented by the `CXXThisExpr`. This is converted in IR into: *this->val*.

The compound assignment operator used in `this->val += diff;` is represented by the `CompoundAssignOperator` node. In IR this is rewritten into *this->val := this->val + diff*, as presented previously in Subsection 3.3.5.

In Listing 3.37 the resulting INSPIRE implementation of the `Counter::inc` member function is presented.

The IR *function type* would be similar to *Counter :: (int⟨4⟩) → unit*. Note that the parameters of the IR *function*, for the member function, is extended to represent the (implicit) `this`-object.

As detailed in Section 3.3.2 during the conversion process we store the converted IR for functions and types in an IRTranslationUnit along with an IR symbol by which they are used. In this case let *inc* be the symbol. This is used in Listing 3.39.

```
(ref<Counter> this, int<4> diff) -> unit {
    this->val := this->val + diff;
};
```

Listing 3.37: IR of `Counter::inc` member function of the `Counter` class

To finish the `Counter` example, we show in Listing 3.38 a simple `main` function (the entry point), instantiating a `Counter` object and calling the `inc` member function one.

```
int main() {
  Counter c;
  c.inc();
  return 0;
}
```

Listing 3.38: C++ `main` function using the `Counter` object

The resulting IR for the `main` function is shown in Listing 3.39. The default argument for the `diff` parameter of the `inc` member function is provided as an explicit argument in the call. Similarly the `this`-object is provided as an explicit argument. Again this IR implementation is stored in the IRTranslationUnit. Starting from the final IRTranslationUnit the resulting IRProgram is resolved. This means that every IR symbol is replaced by the implementation it represents. More details on this resolution is given in Section 3.4.

```
fun() -> int<4> {
  decl ref<Counter> cObj =
    Counter_Ctor( var(undefined(type<Counter>)), 0);
  inc(cObj, 1);
  return 0;
};
```

Listing 3.39: IR for `main` function given in Listing 3.38

3.4 IRTranslationUnit

3.4.1 Overview

In the IRTranslationUnit we collect all the converted IR generated by the Converters, be it for functions, types or global variables. Basically it is the IR equivalent of one Clang AST `TranslationUnitDecl` node. During the Frontend we use a symbolic form of INSPIRE to avoid resolving recursive types and functions already during the conversion step from the Clang AST to INSPIRE. This means that every (user-defined) type gets a symbolic IR type (i.e. an abstract type) which is used at every occurrence instead of the actual IR type. The same holds for functions where a function symbol (i.e. an abstract function) is used at every call to that function instead of the actual IR *lambda expression* representing the functions implementation. After merging the IRTranslationUnits into one single IRTranslationUnit the symbolic IR has to be resolved to get the final IRProgram. The resolution of the symbolic IR is handled by the Resolver, which resolves the possibly recursive types and functions and replaces the IR symbols by their actual IR implementation.

In Figure 3.7 we give a short overview of the process from converting TranslationUnits into IRTranslationUnits, merging them into a single IRTranslationUnit and resolving it into an IRProgram.

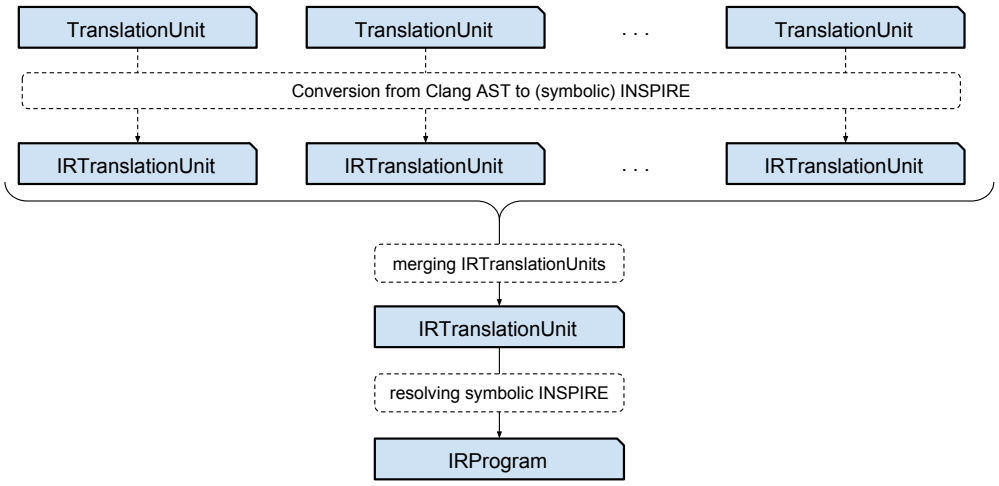

Figure 3.7: Merging multiple IRTranslationUnits and resolving the symbolic INSPIRE into an IRProgram

3.4.2 Structure and merging of IRTranslationUnit

In the IRTranslationUnit we use different containers to store the various generated IR for types, functions, global variables.

Types All the generated types are stored in a map. The IR symbol is the key and the actual implementation is the value of this type map.

Functions All the generated functions are stored in a map. The IR symbol is the key and the actual definition of the function is the value of the function map.

Global variables All the generated global variables and their initialization are stored as a list of pairs. The pair represents the IR symbol for the global variable, and the initialization expression.

ClassMetaInfo During the conversion process we store the *ClassMetaInfo* annotation in the IRTranslationUnit. The *ClassMetaInfo* annotation is attached to the class type in the final resolution step.

Merging IRTranslationUnits The merging of several IRTranslationUnits consists of pairwise copying together each of the containers for types, functions, global variables, and ClassMetaInfo.

The Figure 3.7 pictures conversion of multiple TranslationUnits into their corresponding IRTranslationUnits, and the merging of multiple IRTranslationUnits into a one single final IRTranslationUnit from which the resulting IRProgram is resolved.

3.4.3 Converting an IRTranslationUnit into an IRProgram

After the multiple IRTranslationUnits got merged into a single IRTranslationUnit we need to convert it into an IRProgram. This conversion consists of three steps:

- resolve symbolic IR of the entry point
- resolve symbolic IR of the *ClassMetaInfo* annotations
- add global variable initializers to the entry point

Resolving the symbolic IR is handled by the Resolver, details are provided in the following section.

The initialization of global variables is represented as assignment statements (i.e. *globalVar = initExpr;*). These initializations are prepended to the function body of the entry point, typically the `main` function.

Resolver

The Resolver is responsible for resolving and replacing every IR symbol found in a given IR node. Every IR symbol is replaced by the actual definition it represents after resolving possibly recursive types and functions.

For IR symbols which represent a recursive type or recursive function the Resolver introduces an IR *recursive type* or *recursive function*. An IR symbol is direct recursive if it is used in the definition it is associated with, or mutual recursive if two or more symbols are used mutually in their definitions.

In Listing 3.46 and 3.48 we give examples for direct and mutually recursive functions, and in Listing 3.42 and 3.44 we give examples for direct and mutually recursive types in symbolic IR.

After the Resolver is finished, the IR does not contain any IR symbols anymore. Every IR symbol is replaced by the definition it represented. The self-contained structure of INSPIRE is now established.

Resolving algorithm The resolving algorithm used by the Resolver acts on an IR node. Such an IR node can be a symbol or a definition of an IR symbol.

1. Collect set of contained symbols – First the Resolver collects an initial set of contained symbols. If the starting node was an IR symbol the set contains only one element: the given IR symbol. If the starting node was a definition of a type or function the Resolver visits all nodes depth-first and collects all used IR symbols.

2. Build dependency graph – Starting from the initial set the Resolver builds a dependency graph. The dependency graph's nodes are symbols, and the vertices are dependencies between symbols. Such a dependency exists between a symbol S_a and the symbols contained in the definition associated with symbol S_a.

3. Compute strongly connected components (SCCs) graph – From the dependency graph build up in step two the SCCs are computed. This is done by the `Boost::graph` library [16].

4. Resolve SCCs components bottom up – The SCCs are sorted in reverse topological order to resolve the dependency graphs SCCs from bottom up. The topological sorting is done by `Boost::graph` library [16].

Components with one node need to be checked if they are recursivly dependent on themselves. That means their definition contains the symbol.

Components with more than one node are mutually recursive, thus all the nodes are put into a recursive type/function.

- For recursive types a new IR *recursive type* is added. This IR *recursive type* consists of an IR *type variable* and its definition for every component representing a type of the SCC graph.
- For recursive functions a new IR *recursive function* is added. This IR *recursive function* consists of an IR *variable* and its definition put in a IR *lambda expression*.

5. Resolve input – Finally all the IR symbols are replaced by the resolved implementations.

3.4.4 Examples

Following are several examples how IRTranslationUnits are resolved into IRPrograms.

Simple Example

In the simple example several functions are called from each other. In the IRTranslationUnit the functions are represented by their IR symbol. After resolving the IRTranslationUnit, the symbols got replaced by their corresponding implementation.

```
1 Types:
2     int : int<4>;
3 Functions:
4     func : () -> int { return 1+2; }
5     func1 : () -> int { int x = func() + func(); return
          x; }
6     main : () -> int { return func1(); }
7 EntryPoints:
8     main
```

Listing 3.40: IRTranslationUnit example

```
() -> int<4>
{
    return (() -> int<4>
      {
    int<4> x =
        (() -> int<4> { return 1+2; }) +
        (() -> int<4> { return 1+2; });
    return x;
      }
}
```

Listing 3.41: IRProgram resolved from the IRTranslationUnit example given in Listing 3.40

Recursive type examples

For the resolution of recursive types we give two examples. One with a direct recursion and another one being mutual recursive over two types. Before resolving the IRTranslationUnit the recursive types are IR *struct* types using IR symbols for their members. After the resolution the IR *struct* types turned into IR *recursive* types.

Direct recursion In the Listing 3.42 the symbol `List` is used in line 3 of the definition of the `List` type.

```
List: struct List {
    int data;
    ref<array<List,1>> next;
  }
```

Listing 3.42: Direct recursive type example (symbolic IR)

```
let type0 = struct 'List <
  data:int,
  next:ref<array<'List,1>>
>;
let type1 = rec 'List {
  'List = type0
};
```

Listing 3.43: Direct recursive type example (resolved IR)

Mutual recursion In the Listing 3.44 the symbols `Inner`, and `Outer` are used in line 3 and line 7 of the definitions of the `Inner` and `Outer` types.

```
1 Inner : struct Inner {
2     int i;
3     ref<array<Outer,1>> outer;
4   }
5 Outer : struct Outer {
6     int o;
7     ref<array<Inner,1>> inner;
8   }
```

Listing 3.44: Mutual recursive type example (symbolic IR)

```
1
2 let type0 = struct 'Inner <
3   i:int,
4   outer:ref<array<'Outer,1>>
5 >;
6 let type1 = struct 'Outer <
7   o:int,
8   inner:ref<array<'Inner ,1>>
9 >;
10
11 let type2 = rec 'outer {
12   'inner = type0;
13   'outer = type1;
14 };
15 let type3 = rec 'inner {
16   'inner = type0;
17   'outer = type1;
18 };
```

Listing 3.45: Mutual recursive type example (resolved IR)

Recursive function examples

For the resolution of recursive functions we provide two examples. One with a direct recursion and another one being mutual recursive over two functions. Before resolving the IRTranslationUnit the functions use symbols for the recursive calls. After resolution the functions are represented as IR *recursive function*.

Direct recursion The symbol `succ` is used in line 5 of the definition of the successor function.

```
1 succ : (int n) -> int {
2   if(n == 0) {
3     return 1;
4   } else  {
5     return succ(n-1) + 1;
6   };
7   }
```

Listing 3.46: Direct recursive function example (symbolic IR)

```
1 let fun0 = recFun v1 {
2   v1 = fun(int v0) -> int {
3     if(v0==0) {
4       return 1;
5     } else {
6       return v1(v0-1);
7     }
8   };
9 };
```

Listing 3.47: Direct recursive function example (resolved IR)

Mutual recursion The symbols `even` and `odd` are used in line 5 of the definition of the even function and in line 12 of the definition of the odd function.

```
even : (int n) -> bool {
  if(n == 0) {
    return true;
  } else {
    return odd(n-1) ;
  };
  }
odd : (int n) -> bool {
  if(n == 0) {
    return false;
  } else {
    return even(n-1) ;
  };
  }
```

Listing 3.48: Mutual recursive function example (symbolic IR)

```
let fun0 = recFun v10 {
  v10 = fun(int v0) -> int {
    if(v0==0) {
      return true;
    } else {
      return v11(v0-1);
    }
  };
  v11 = fun(int v1) -> int {
    if(v1==0) {
      return false;
    } else {
      return v10(v1-1);
    }
  };
};

let fun1 = recFun v11 {
  v10 = fun(int v0) -> int {
    if(v0==0) {
      return true;
    } else {
      return v11(v0-1);
    }
  };
  v11 = fun(int v1) -> int {
    if(v1==0) {
      return false;
    } else {
      return v10(v1-1);
    }
  };
};
```

Listing 3.49: Mutual recursive function example (resolved IR)

Chapter 4

Supporting Languages and API Constructs

4.1 Overview

In Chapter 3 we discussed the conversion process from the Clang AST to INSPIRE for C/C++. In this chapter we discuss how this conversion process can be altered with the help of the plugin system and how different utilisations of the plugin system can be used for:

- adding support for additional sequential language standards such as C++11
- interfacing sequential third-party libraries such as Boost and the C++ STL
- supporting parallel languages and APIs e.g. OpenMP, CILK, MPI, or OpenCL

First in Section 4.2 we give a short introduction into the plugin system, providing the means to extend and alter the conversion process to support the various languages and APIs. More details on the plugin system, and its implementation can be found in Stefan Moosbruggers master thesis [11].

Then we discuss in Section 4.3 how we support additional sequential language standards, in particular we give details on the support for C++11.

Following in Section 4.4 we detail how we interface sequential APIs, such as C++ STL, and Boost. Especially for C++ STL we provide a detailed example.

Finally in section 4.5 we outline the Insieme Frontend support for OpenMP, MPI, CILK and OpenCL, which can serve as an example for other C/C++ based parallel languages and APIs.

4.2 Plugin System

4.2.1 Overview

This section gives an introduction on the plugin system and its different phases, which are pictured in Figure 4.1. It is based on the thesis of Stefan Moosbrugger [11], which provides a more complete introduction and explanation.

The Plugin system provides means to alter and extend the standard conversion process presented in Chapter 3 at various phases. These phases have a fixed order:

1. Clang frontend phase – influence how the Clang is setup to parse the input source code
2. Conversion phase – alter and extend how certain Clang AST nodes are converted into IR
3. Post conversion phase – alter the already generated IR of Clang AST nodes
4. IR (IRTranslationUnit) phase – alter and transform the IR stored in an IRTranslationUnit
5. IR (IRProgram) phase – alter and transform the IR stored in an IRProgram

The Pragma handling is also implemented via the plugin system. It defines which pragmas are handled and how they are handled. The handling of the pragmas itself is not fixed to a certain phase in the conversion process as it might be necessary to influence different phases.

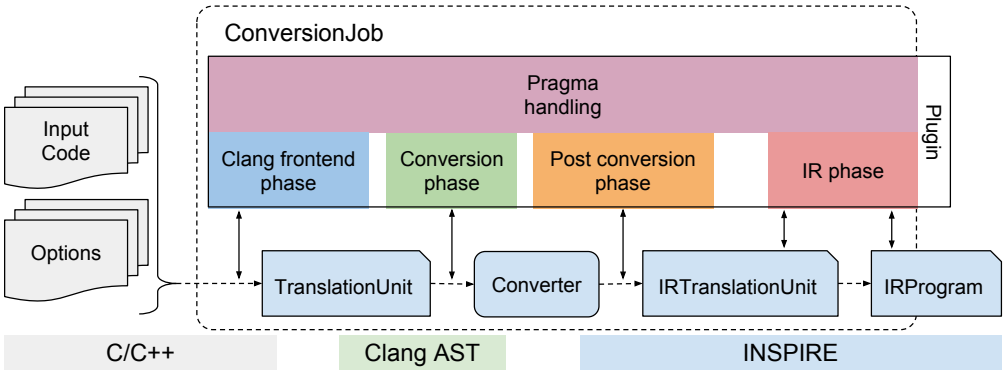

Figure 4.1: Overview of a plugin and its phases

4.2.2 Clang frontend phase

The Clang frontend phase allows the plugin to alter and modify the setup of the Clang infrastructure. This includes for example registering the pragma handlers provided by the user, adding special include paths, or setting macro definitions for the preprocessor.

Kidnapped header The plugin user can provide a custom version of a header file which would be found otherwise in a different include path. This gives the option to kidnap header files by adding an include header search path in front of all other header search path. Thus when the Clang Preprocessor searches the header files the custom version is found first.

Injected header The plugin user can specify header files to *inject*. This means an additional `#include` directive is prepended at the beginning of the input file. This happens before preprocessing the input file.

Macro definitions The plugin user can define macros which are then preprocessed by the Clang preprocessor in the parsing step.

4.2.3 Conversion phase

The Conversion phase offers the possibility to modify and alter the way Clang AST nodes are converted, instead of converting them in the standard conversion process as presented in Chapter 3.

For any given instance of a Clang AST node only one plugin or the standard conversion process can be used to convert that specific node into IR. In other words, for the same Clang AST node type there can be several implementations how to convert these nodes into IR, but only one of these implementations is used. The plugin system offers method to alter the conversion of the basic node types i.e. `Expr`, `Stmt`, `Type`, and `Decl` nodes.

4.2.4 Post conversion phase

The Post conversion phase offers the possibility to modify and alter the generated IR of a specific Clang AST node type, previously converted from the Clang AST. The IR to be modified can be generated by the standard conversion process or a plugin (handled in the Conversion phase). The plugin system offers method to alter the generated IR of the basic node types i.e. `Expr`, `Stmt`, `Type`, and `Decl` nodes.

4.2.5 IR phase

In the IR phase one can alter and modify the generated IR on the level of a single IRTranslationUnit or on the level of an IRProgram (i.e. after merging all IRTranslationUnits and resolving the IR symbols).

Modifying an IRTranslationUnit After the Clang AST was converted into IR, it is stored in an IRTranslationUnit corresponding to the Clang AST. This phase offers the possibilities to modify the contents of that IRTranslationUnit. The plugin user can alter specifically the stored types, functions, global variables and their initializers, or any of the ClassMetaInfo.

Modifying an IRProgram After merging all IRTranslationUnits together and resolving the symbolic IR, the IRProgram is available. This phase offers the possibility to modify the final IRProgram. To navigate the IRProgram the user can use the tools provided by the Insieme project to analyse and transform INSPIRE.

4.2.6 Pragma handling

To implement and provide custom pragma handling the pragma plugin needs to be able to modify and interact with the conversion process at all the different phases. With the pragma handling implemented in the plugin system, the user has a flexible way to add user-defined pragmas and also add user-defined handling of pragmas without changing the standard way of the conversion process.

Recognizing user-defined pragmas

Modifies mainly at the Clang frontend phase, when preprocessing the input source. This can be used to specify custom pragmas by the user.

User-defined handling of pragmas

This defines how the generated IR nodes are related with a pragma. For example adding to an IR node an annotation with additional information. Moreover it defines how to handle the IR nodes related with the pragma.

4.3 Supporting sequential Language Standards

4.3.1 Overview

As already explained in Chapter 3 the Insieme Frontend is based on the Clang AST, thus we can only support languages supported by the Clang project. This includes the languages, as well as certain standards or features of the languages.

Such an additional language standard is the C++11 standard. The Clang AST provides support for it and the conversion of the C++11 specific nodes is implemented in Insieme as a plugin. The following subsection exemplifies how a conversion for such an additional language standard, such as C++11, can be implemented.

A similar procedure as presented, implementing the conversion in a plugin, can be applied to other language standards, or specific language features. In that way the code base is separated and its maintainability is increased.

4.3.2 C++11 Plugin

In order to support new C++11 features, several nodes were added to the Clang AST:

- `AutoType` - the `auto` type
- `DecltypeType` - the `decltype` type
- `CXXForRangeStmt` - represents a range-based for-loop
- `CXXNullPtrLiteralExpr` - the `nullptr` literal
- `LambdaExpr` - models a lambda expression, the resulting function and possible captured variables

The conversion of these nodes, from Clang AST into IR, is implemented in a plugin. This plugin handles these nodes in the Conversion phase. By implementing the conversion of these C++11 specific nodes in a plugin, we increase the maintainability of the standard conversion process and get the possibility to switch between different language standards.

`AutoType`

The `auto` type automatically deduces the type of a variable, that is being declared, from its initializer. For the `auto` type node the Clang AST provides already the underlying type. Thus we refer the conversion to the TypeConverter introduced in Chapter 3.

`DecltypeType`

The `decltype` specifier can be used in declarations and gives the declared type of an entity or the return type of an expression. Thus `decltype(E)` returns the type of the entity or expression `E`.

The `DecltypeType` node provides as a child node, the expression or entity as an `Expr` node. Thus we use the ExprConverter (as introduced in Chapter 3) to convert the expression into IR. From the resulting IR representation we can query the IR type.

`CXXForRangeStmt`

The `CXXForRangeStmt` node represents a range-based for-loop. As this is not a count-controlled for-loop we turn it into an IR while-loop. The `CXXForRangeStmt` node provides with different child nodes the necessary loop properties such as: the loop-variable, the condition expression, the increment expression, the begin and end of the range, and the loop body. These child nodes are converted into IR by the StmtConverter or ExprConverter introduced in Chapter 3, then put together to the resulting IR while-loop.

`CXXNullPtrLiteralExpr`

The C++11 `nullptr` literal is an explicit null pointer. The `CXXNullPtrLiteralExpr` node provides the underlying type, which is converted with the TypeConverter introduced in Chapter 3. The resulting IR type is used to generate a call to the IR *RefReinterpret* operator to represent the `nullptr` as a correctly typed IR *RefNull* expression.

`LambdaExpr`

The Clang AST `LambdaEpxr` node models a C++11 lambda expression. This Clang AST node provides as children the resulting C++11 lambda function declaration and possible captured variables.

The implementation of a C++11 lambda function is typically done as a small anonymous class, where the call operator (i.e. `operator()`) is overloaded with the C++11 lambda function and the captured variables are passed to the constructor of the anonymous class, to initialize member variables to be used in the `operator()`.

As the Clang AST does not use identifiers for the member fields of anonymous classes, we enumerate the member fields in the resulting IR `struct` type representing the anonymous class. During the conversion of the C++11 lambda function declaration the captured variables are turned into IR *variables* which

need to be replaced later on by the access to the member field of the anonymous class.

The conversion of a `LambdaEpxr` node is handled in two steps:

1. First the `LambdaEpxr` node itself is converted. This is done during the Conversion phase.

   We begin the conversion by converting the captured variables into IR *variables*.

   Next the C++11 lambda function declaration representing the C++11 lambda function is converted. This generates an IR symbol for the C++11 lambda function and an IR *function*. The previously converted IR *variables*, for the captured variables, are used inside the resulting IR *function* representing the C++11 lambda function. The IR symbol is used at every callsite and the IR *function* is stored in the IRTranslationUnit as explained before with functions and member functions.

   Furthermore the captured variables are used as arguments for the constructor of the anonymous class to initialize the member fields.

2. To be able to correctly access the member fields of the `this`-object of the anonymous class type, we need to fix the member access. This is done in the Post-clang phase. We replace all the IR *variables* representing the captured variables by the correct member access to the `this`-object.

The call to such a C++11 lambda function itself is handled in the standard conversion process, as the Clang AST represents it as a `CXXOperatorCallExpr` with the variable representing the lambda expression as the `this`-argument, and an overloaded call operator (i.e. `operator()`) as the lambda function.

Listing 4.1 gives a simple example for a C++11 lambda function, and Listing 4.2 the shows the exemplary anonymous class used to implement it. Listing 4.3 shows the simplified IR (some details are omitted, and the naming scheme of types a functions may differ).

```
1 void main() {
2   int x = 10;
3   auto lambda = [x]() { std::cout << x << std::endl; };
4   lambda();
5 }
```

Listing 4.1: C++ lambda function example

```
void main() {
  int x = 10;
  struct ANONYMOUS {
    ANONYMOUS(int _x) : x(_x)
    void operator() { std::cout << this->x << std::endl; }
  private:
    int x;
  };
  ANONYMOUS(x).operator();
}
```

Listing 4.2: C++ lambda function example with examplary anonymous class

```
let ANONYMOUS = struct<
  __m0:int<4>
>;

let operator() = mfun ANONYMOUS v0 :: () unit { std::
    cout << v0->__m0 << std::endl; }

...

let main = fun() -> unit {
  decl ref<int<4>> v1 = var(10);
  decl ref<ANONYMOUS> v2 = ANONYMOUS(v1);
  fun000(v2);
}
```

Listing 4.3: C++ lambda function example in simplified IR

4.4 Supporting sequential APIs and Third-party Libraries

4.4.1 Overview

In order to be able to support and interface an external third-party library, we utilize INSPIREs abstract (generic) types and functions. To generate those IR types, literal and functions we need a way to identify calls and usages to the library's functions and types and *intercept* their conversion. Furthermore we need to provide information about the headers of the library. In the Insieme Backend we then generate the corresponding include directives and calls to interface the external libraries. Due to intercepting these third-party library functions and types we avoid to recompile the library.

In short we need three steps: (1) identify functions and types of external

libraries to intercept, (2) provide abstract IR function and types for them, and (3) provide information about the header of the libraries.

These steps are implemented in a plugin and a utility:

- The interception plugin is handling step one, it utilizes the Interceptor to identify calls and usages to functions and types to intercept.

- The Interceptor is handling steps two, and three, it is a utility to identify functions and types to intercept and generate the abstract (generic) IR types and function for them.

First we provide details on the plugin and the utility and then we discuss the setup of interception for third-party libraries using the C++ Standard Template Library as an example.

4.4.2 The Interception Plugin

The interception plugin is only used to identify the Clang AST nodes which are referring or using functions and types which need to be intercepted. The actual interception is handled by the Interceptor. We intercept the conversion of functions already when the `FunctionDecl` node is converted. For functions, to intercepted we hand over the `FunctionDecl` to the Interceptor and get back the intercepted abstract (generic) IR function.

Similarly for possibly intercepted types, we need to check every `Type` node. If the type needs to be intercepted, the interception is handled by the Interceptor. The `Type` node is handed to the Interceptor and we get back an abstract (generic) IR type.

In that way we provide for the rest of the conversion process already the intercepted abstract (generic) IR functions and types. Thus whenever an intercepted `FunctionDecl` or `Type` node is encountered the abstract (generic) IR function or type is used. In other words, when the `CallExpr` to an intercepted function is converted, the callee (i.e. the intercepted `FunctionDecl`) is already converted into the abstract (generic) IR and stored in the IRTranslationUnit. The `CallExpr` conversion uses the IR *symbol* associated with the function, similarly as in the non-intercepted case as presented in Section 3.3.2.

In some cases, it might be necessary to provide more information about the call or usage itself, to the Interceptor. For example templated functions might be called with explicit template arguments. Such information is provided by the Clang AST via the `DeclRefExpr` node.

4.4.3 The Interceptor

The Interceptor is our utility to identify if a callee, i.e. a function, or a type needs to be intercepted and generates the corresponding abstract (generic) IR.

For intercepted functions we introduce an abstract (generic) IR function, representing the callee, and for intercepted types we introduce an abstract (generic) IR type, representing the used type. Furthermore the Interceptor attaches an *IncludeAnnotation* to the generated IR, providing the header information for the intercepted library.

Identify functions and types to intercept

In order to be able to decide if a function or a type needs to be intercepted we use its qualified name, i.e. the name of the function or type and all its enclosing namespaces. To intercept a certain name or namespace, a regular expression, matching that name or namespace, needs to be provided to the Interceptor. If this regular expression matches the qualified name, the Interceptor generates the according abstract IR function or type. In order to intercept multiple namespaces or names the regular expressions are concatenated with a logical-or.

In addition to the regular expression, for matching the (qualified) names, the include path to the headers of the intercepted library needs to be provided to be able to attach the correct *IncludeAnnotation*.

Intercepting functions

To intercept a function we generate abstract (generic) IR literal with the function's type and its qualified name. Then an *IncludeAnnotation* is attached to the literal, specifying the header file where the function is declared in the external library.

In the Insieme Backend we generate from the IR literal a call to the function of the external library. From the attached *IncludeAnnotation* the Insieme Backend adds an `#include` directive referring to the header required for the intercepted function. Hence the generated code links correctly against the intercepted library.

Intercepting types

To intercept a type we generate an abstract (generic) IR type with the qualified name of the type. Primitive types are not intercepted, we only intercept user defined types such as `struct`, `class`, `union`, `enum`. These types are stored in the Clang AST in the `TagType` node. Then an *IncludeAnnotation* is attached

to the literal, specifying the header file where the function is declared in the external library.

The Insieme Backend generates from the IR type the according type provided by the external library. From the attached *IncludeAnnotation* the Backend adds an `#include` directive referring to the header needed for the intercepted function. Hence the generated code links correctly against the intercepted library.

For templated (generic) C++ types we utilize the generic type system of the IR and represent them as abstract (generic) IR types.

4.4.4 Intercepting Third-party libraries

In order to intercept a third-party library the user needs to provide a regular expression matching the qualified name of the functions and types to intercept. Besides that informations about the headers of the library need to be provided.

For example in order to intercept the Boost library one could use the regular expression `boost::.*`. As header information the path to the Boost headers directory is needed.

Intercepting C++ Standard Template Library Now a short example on how we intercept the C++ Standard Template Library (STL). More precisely we give an example in which we intercept the usage of a `std::vector` type and its operators.

The regular expression used to identify types and functions of the STL is rather simple. All STL types and functions are located in the `std` namespace, hence we use `std::.*` as regular expression.

When the Insieme Frontend is setup to convert C++ input codes, the interception of the STL is setup by default. The regular expression and the necessary header information are preset during the setup of the ConversionJob. The header information for the STL is queried during setup from the third-party backend compiler, typically GCC.

In Listing 4.4 we give a short example C++ code where an empty `std::vector` - instantiated to an integer i.e. `std::vector<int>` - is declared in line 1. In line 2 we add an element (i.e. `5`) at the end of the vector with the `push_back` operation. Listing 4.5 then shows the resulting IR for the code given in Listing 4.4.

The templated (generic) C++ type `std::vector<T>` is represented in IR as an abstract (generic) type $\textit{std::vector}\langle\alpha\rangle$. This templated C++ type needs a proper instantiation when used in an actual instance. In this example we use a simple vector of integer, thus the C++ vector type is instantiated to an integer `std::vector<int>` which is represented in IR as $\textit{std::vector}\langle\textit{int}\langle 4\rangle\rangle$.

```
1 std::vector<int> v; //implict constructor call
2 v.push_back(5);
```

Listing 4.4: C++ std::vector

```
1 ref<std::vector<int<4>> v = std::vector(...); //ctor-
    literal
2 push_back(v, 5);
```

Listing 4.5: Intercepted std::vector

In Listing 4.4 we use two operations of the C++ vector, the default constructor and the `push_back` operator, these are represented as two abstract (generic) IR operators. The IR type variable α, it is used in these two operations. The IR type variable is replaced by the actual type used in the instance (i.e. $int\langle 4 \rangle$).

- The default constructor is represented as an IR literal with an IR (constructor) function type: *std::vector : std::vector*$\langle\alpha\rangle$ *:: (...)*
- The `push_back` operator is represented as an IR literal with the (member) function type : *push_back : std::vector*$\langle\alpha\rangle$ *::* $(\alpha) \rightarrow$ *unit*

4.5 Support for parallel APIs

4.5.1 Overview

This section gives a short overview on how we support parallel APIs and languages in the Insieme Frontend. Typically the implementation is done as a plugin, which provides the means to convert the given parallel API and language. The following subsections can serve as an example on how to implement other C/C++ based parallel languages and APIs.

The details on how the different parallel models are represented in INSPIRE are *not* subject of this section. An overview can be found in the dissertation of Herbert Jordan [9].

4.5.2 OpenMP

The support for OpenMP is implemented as a plugin which provides the necessary pragma handlers for the various OpenMP pragmas. For every OpenMP pragma we want to be able to support a pragma handler of its own is provided. The pragma handlers convert the given pragma, and the statements and expressions it is related to, into suitable IR constructs modeling the semantics of

the OpenMP pragma. For details on the modeling of the various pragmas in INSPIRE see the dissertation of Peter Thoman [10].

4.5.3 CILK

CILK is a language standard extending C/C++ by introducing new keywords. The essential ones are `spawn` and `sync`, modeling a fork-join parallelism. As Clang is currently not supporting CILK, the Insieme Frontend needs a trick to parse and convert CILK source into INSPIRE.

We setup the Clang preprocessor in such way that the keywords are interpreted as macros and replaced by a pragma definition during the preprocessing. For example the keyword `spawn` is turned into a pragma with a macro definition (`"spawn = _Pragma(cilk spawn)`).

After preprocessing a CILK input code, with the additionally provided macros, every statement containing a cilk keyword is marked with an additional pragma. Later in the conversion process the CILK pragma handler converts the marked statements into the IR constructs used to represent the fork-join parallelism.

4.5.4 MPI

MPI is a standard providing an API and a library for message passing. Thus in order to analyse MPI we need to identify calls to the MPI. These calls are recognized as the name of the called function has `MPI_.*` as prefix.

The identification of Clang AST `CallExpr` nodes with a callee to an MPI function with MPI as prefix is done as a plugin. The callees are than converted into specific IR operators modelling the MPI calls.

4.5.5 OpenCL

OpenCL provides its source code in two parts, a host part and a kernel part. The conversion of these two parts is handled by two plugins, each handling respectively one part.

Typically the kernel source is meant to be run on a *device* (e.g. a GPU, or an accelerator card). The host code is responsible for the set-up of the kernel as well as providing the data the kernel should operate on. This *boilerplate* code is written using the OpenCL API with library calls. During the conversion of OpenCL host and kernel code the *boilerplate* code, needed for the setup of devices and kernels, and the kernels invocation, needs to be identified. These OpenCL library calls are not represented in the resulting IR so as to generate IR code focused on the essential kernel calls and data manipulation operations.

In the Insieme Backend this *boilerplate* code is re-added where necessary with appropriate calls to the Insieme Runtime System.

In order to identify the kernel code, a pragma needs to be added by the programmer to mark the kernel code. When a OpenCL kernel code is encountered it is converted into a ConversionJob of its own. This ConversionJob is set-up with the OpenCL kernel plugin enabled. After the conversion of the kernel is finished, it gets inlined into the IR of the host code. As there could be multiple kernels this is done for every marked kernel file.

Chapter 5

Conclusions and Future Work

Developing parallel programs which utilize the available parallel hardware efficiently is complex and time consuming. The software developer needs to manage a lot of different parallel programming standards and APIs in order to reveal and leverage parallelism. To handle the complexity of tuning and developing an efficient parallel program the software developer needs tool support. One such tool would be a compiler. The goal of the Insieme project and its Insieme Compiler is to offer such tool support to the software developer. A typical compiler uses an intermediate representation to analyse and optimize programs. This reduces the level of abstraction as the input code it is translated into a lower-level intermediate representation. This conversion of the input code from a higher-level language to the lower-level intermediate representation is done by the compiler's frontend.

In the context of the Insieme Compiler, the Insieme Frontend is responsible for translating the input code given in a higher-level language into INSPIRE. As part of this master thesis an existing frontend capable of converting C into INSPIRE was extended to support C++. By extending the Insieme Frontends language support with support for C++, the Insieme project is able to offer its existing optimizations to C++ programs and lays the ground to research optimizations specific to C++.

Developing a frontend from scratch is challenging, tedious, and error-prone, especially for a complex language like C++. Hence the Insieme Frontend relies for parsing and generating an abstract syntax tree on the Clang project. The Clang project offers a product-quality frontend for the LLVM compiler project. It offers support for the C language family, this includes C and C++ amongst others. The Clang project uses a detailed and expressive abstract syntax tree. The Insieme Frontend utilizes the Clang project to generate from an given input code a Clang AST and convert the Clang AST into INSPIRE. Thus the Insieme Frontend is able to translate C and C++ programs to INSPIRE.

5.1 Contributions

On of the main contributions of the author is this master thesis itself, as it acts as a documentation for the Insieme Frontend. It documents the details of the conversion of input code into a Clang AST and further into INSPIRE. Starting from the basics on how multiple translation units are handled, and how the Clang AST is traversed, to details on the conversion of specific Clang AST nodes into a semantical equivalent in INSPIRE.

Minor contributions in the general development of the Insieme Frontend involved:

- improvements of the include-path setup of the Clang infrastructure used for parsing C/C++ input codes
- handling of *ClassMetaInformation* annotation for C++ member functions
- support for single instruction multiple data (SIMD) operators for vector types
- improvements and bug fixes of global variable support

Besides these minor contributions, major contributions are listed below.

C++ Language Support Further contributions in course of this master thesis were made by extending an existing C frontend with support for C++. This included amongst other, the implementation of the conversion of the following C++ features respectively C++ specific AST nodes:

- Exception handling – implementing the conversion of the AST nodes: `CXXTryStmt`, `CXXCatchStmt`, `CXXThrowExpr`
- Heap memory allocation and deallocation – implementing the conversion of the AST nodes: `CXXNewExpr`, `CXXDeleteExpr`
- Handling of overloaded C++ operators – implementing the conversion of the `CXXOperatorCallExpr`
- C++ "named" casts – `dynamic_cast`, `static_cast`, `const_cast`, `reinterpret_cast`
- Conversion of templated types and functions
- Conversion and handling of C++ constructor and initializer

Support for Third-Party Libraries Another major contribution is the Interceptor. With the Interceptor the Insieme Frontend is able to *intercept* the conversion of external third-party libraries. Thus the Insieme Compiler is able to interface such external libraries without the need of recompiling these libraries.

5.2 Future Work

The Insieme project and its compiler is continuously developed and thus new features and applications are constantly developed. In the following we list possible future work in the domain of the Insieme Frontend which improves the maintainability of the frontend, the language support, or opens up possible research field.

Language standard support Important future work is the addition of upcoming language standards of C++ i.e. C++14, and improvement of the currently partial support for the C++11 language standard. An interesting aspect is the extension of the C++ standard library with functions and types for concurrent and parallel programs. It might be beneficial to recognize these concurrent and parallel functions and represent their semantics with fitting parallel INSPIRE constructs. These extensions can be implemented as plugins utilizing the frontend plugin system.

Merging Converter Currently the Insieme Frontend organizes the conversion of input codes in two different converters. One taking care of conversion of Clang AST nodes used by C and C++, and another one handling C++ specific nodes. It might improve the maintainability of the code by merging the converters into one converter handling nodes used for C and C++, and using the frontend plugin system to handle all C++ specific nodes.

Annotating intercepted containers with ADTs Currently intercepted functions and types are seen as black box. It might be beneficial to provide additional information about the intercepted types and functions. For example the intercepted types, for STL containers and their accompanying functions, could be recognize and annotated with the corresponding abstract data types (ADT).

List of Figures

2.1 Typical setup of the Insieme infrastructure 3

3.1 Overview of the Insieme Compiler 11
3.2 Processing a ConversionJob 12
3.3 Structures of different representations 14
3.4 Interaction between the main frontend components 21
3.5 Traversing a TranslationUnitDecl 22
3.6 Converting a `FunctionDecl` 25
3.7 Merging multiple IRTranslationUnits and resolving the symbolic INSPIRE into an IRProgram 54

4.1 Overview of a plugin and its phases 64

List of Tables

2.1 Operators of ref extension 7
2.2 Operators on *array* container 8

3.1 IR representations of pointer types 31

List of Listings

3.1 C function with a Call 19
3.2 Simplified AST of example given in listing 3.1 20
3.3 Templated C++ function 27
3.4 IR struct for templated C++ function 27
3.5 C/C++ function pointer type example 30
3.6 Simple recursive C/C++ `struct` type 32
3.7 IRTranslationUnit entry of a simple recursive *struct* type 32
3.8 IR construct for enum constant 32
3.9 IR construct for enum type 33
3.10 Simple templated C++ struct 33
3.11 IR struct for simple templated C++ struct 33
3.12 IR construct for C++ reference type 33
3.13 IR construct for C++ const reference type 34
3.14 If-statement in C/C++ 35
3.15 If-statement from Listing 3.14 in IR 35
3.16 C/C++ For-Loop which gets rewritten into While-Loop 35
3.17 Rewritten C/C++ While-loop from Listing 3.16 (in IR) 36
3.18 C/C++ While-Loop 36
3.19 While-loop from Listing 3.18 in IR 36
3.20 Switch/Case with fall-through in C 37
3.21 Switch/Case with fall-through in IR 37
3.22 Try-Catch with a catch-all caluse in C++ 38
3.23 Try-Catch with catch-all clause in IR 38
3.24 C function call 43
3.25 Call to `func1` from Listing 3.24 in IR 44
3.26 C++ member function call 44
3.27 Call to `Class::memberFunc` from Listing 3.26 in IR 45
3.28 C++ constructor call 46
3.29 Call to constructor of class `Class` from Listing 3.28 in IR 47
3.30 C function with a Call 50
3.31 Simplified AST of example given in Listing 3.30 50
3.32 IRTranslationUnit resulting from the AST given in Listing 3.31 . 51

3.33 C++ `Counter` class 51
3.34 IR type of the `Counter` class 52
3.35 `Counter::inc` member function of the `Counter` class 52
3.36 Clang AST for member function `Counter::inc` 52
3.37 IR of `Counter::inc` member function of the `Counter` class . . . 53
3.38 C++ `main` function using the `Counter` object 53
3.39 IR for `main` function given in Listing 3.38 53
3.40 IRTranslationUnit example 57
3.41 IRProgram resolved from the IRTranslationUnit example given in Listing 3.40 58
3.42 Direct recursive type example (symbolic IR) 58
3.43 Direct recursive type example (resolved IR) 58
3.44 Mutual recursive type example (symbolic IR) 59
3.45 Mutual recursive type example (resolved IR) 59
3.46 Direct recursive function example (symbolic IR) 60
3.47 Direct recursive function example (resolved IR) 60
3.48 Mutual recursive function example (symbolic IR) 61
3.49 Mutual recursive function example (resolved IR) 62

4.1 C++ lambda function example 69
4.2 C++ lambda function example with examplary anonymous class 70
4.3 C++ lambda function example in simplified IR 70
4.4 C++ std::vector 74
4.5 Intercepted std::vector 74

Bibliography

[1] clang: a c language family frontend for llvm. `http://clang.llvm.org`, June 2014. [Online; accessed 12. November 2014].

[2] Gcc, the gnu compiler collection. `http://gcc.gnu.org`, October 2014. [Online; accessed 12. November 2014].

[3] The llvm compiler infrastructure. `http://www.llvm.org`, June 2014. [Online; accessed 12. November 2014].

[4] Rose compiler infrastructure. `http://rosecompiler.org`, October 2014. [Online; accessed 12. November 2014].

[5] Edg - edison design group. `http://www.edg.com`, October 2014. [Online; accessed 12. November 2014].

[6] Insieme project – mission statement. `http://www.dps.uibk.ac.at/insieme/mission.html`, May 2014. [Online; accessed 12. November 2014].

[7] Insieme project homepage. `http://dps.uibk.ac.at/insieme/`, May 2014. [Online; accessed 12. November 2014].

[8] 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.

[9] Herbert Jordan. *Insieme - A Compiler Infrastructure for Parallel Programs*. PhD thesis, University of Innsbruck, Institue of Computer Science, September 2014.

[10] Peter Thoman. *Insieme RS: A Compiler-supported Parallel Runtime System*. PhD thesis, University of Innsbruck, Institue of Computer Science, July 2013.

[11] Stefan Moosbrugger. *Insieme Frontend Extensions: Plugin System and User Interface*. Master's thesis, University of Innsbruck, Institue of Computer Science, November 2014.

[12] Introduction to the clang ast. http://clang.llvm.org/docs/IntroductionToTheClangAST.html, June 2014. [Online; accessed 12. November 2014].

[13] clang api documentation. http://clang.llvm.org/doxygen, June 2014. [Online; accessed 12. November 2014].

[14] Clang compiler user's manual. http://clang.llvm.org/docs/UsersManual.html, June 2014. [Online; accessed 12. November 2014].

[15] ISO. ISO/IEC 14882:2003: Programming languages — C++.

[16] Jeremy Siek and Lie-Quan Lee and Andrew Lumsdaine. The boost graph library. http://www.boost.org/libs/graph/, June 2001. [Online; accessed 12. November 2014].