Insieme Frontend Extensions: Plugin System and User Interface

Master Thesis in Computer Science

by

Stefan Moosbrugger

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, 6 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.

Stefan Moosbrugger, Innsbruck on the 6 November 2014

Abstract

The Insieme compiler and Insieme Runtime system support the development and optimization of parallel programs. However, the usage of the Insieme infrastructure, especially for larger code bases, is not trivial. Limited software productivity does not only affect the end-user of the Insieme compiler, but also Insieme compiler developers have to be careful when implementing new features, in order to keep the code structure clean, error-free, and readable.

This work aims to develop a high productivity Insieme compiler for both the end-users and the compiler developers. Beside the formal background, the main content of this master thesis describes the implementation of the Insieme frontend and the implementation of the insiemecc driver on the one hand, and a plugin system for the Insieme frontend on the other hand.

The insiemecc driver, provides an easy-to-use interface for the end-users of the Insieme compiler, that can be used as a replacement for GCC, LLVM, or any other kind of C/C++ compiler. The Insieme frontend plugin system, provides an interface for the compiler developers, to modify the standard workflow of the Insieme compiler. This makes it very easy to introduce new functionalities (e.g., implementation of a new language standard) without modifying the core components of the Insieme frontend.

Contents

Chapter 1

Introduction

In the last decade single-core processor systems have been substituted by multi-core processor systems. Because of this fact, it was necessary to change the ways of how computer systems are programmed. The main reasons why multi-core processor systems have become the de facto standard and more important than single-core processor systems are the high power consumption when using high frequencies and the heat dissipation that is difficult to handle. To solve this problems and providing processors with a higher performance, multiple cores are used instead of a single core with a higher frequency.

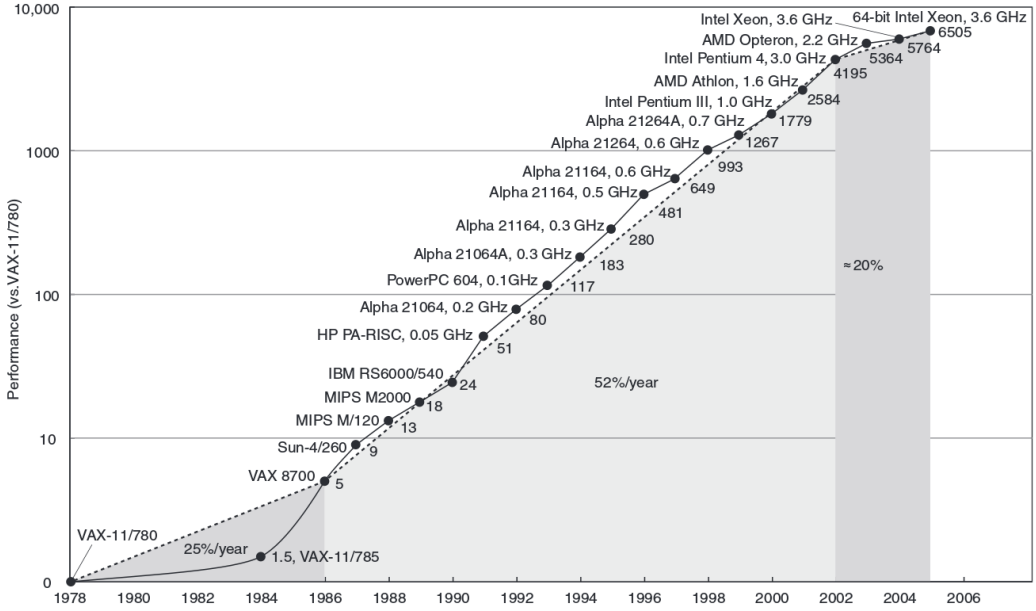

Figure 1.1: Performance history of CPUs [1]

Single-core systems are easy to program and if the frequency of the computing unit is increased the performance will also increase. In simple terms,

this means that an execution that takes one time unit when executed on a 1GHz single-core system will be two times faster when executed on a 2GHz single-core system. This is true for single-core but not for multi-core systems, because the control- and data-management between the computing units have to be considered. The problem when executing software on a multi-core system is that the CPU or the operating system will not take care of the work-load distribution of a single program. This means, if a software programmer is not taking care of the existence of a multi-core system, only one computing unit will be utilized by the software. This can lead to poor and unsatisfying performance results. Figure 1.1 shows the history of CPU performance. The performance increase of more than 50% per year is not valid anymore. Nowadays the performance of a single core is only increasing at about 20% per year, because the chip manufacturers produce multi-core processors instead of faster cores. Therefore it is required to change the way of how software is programmed in order to make use of the chip performance, be energy efficient, save runtime, etc.

There have been several approaches to make it easier for software developers to create software that can be used effectively on multi-core systems. For instance, with OpenMP [2] a programmer can define parallel regions or loops that will be equipped with the correct fork and join calls to execute them in different threads. The problem when using such frameworks or parallel programming paradigms is that the developer has to have special skills, programming knowledge, and when using systems with plenty of cores or heterogeneous environments it gets very difficult to organize the work-distribution in a good way. But not only runtime performance can be important. There can also be objectives like power consumption, efficiency, computing costs, etc. The Insieme project tries to research ways of automatic optimization for such parallel programs.

1.1 My contributions

This master thesis was developed in terms of the Insieme project. There are a lot of people working on the project and as a result of this I will explain my contributions to Insieme:

- Creation of a usable driver: In order to use Insieme effectively, there was a need to create a compiler driver (known as insiemecc) that understands the well known flags that are used in modern compilers (e.g., optimization-, debug-, warning-, include-, linker-flags, etc.). The second feature that was introduced is the creation of object files. A main problem of Insieme

was that the information of all translation units had to be known in order to generate the intermediate representation. This led to a huge memory consumption when using multiple translation units. To get rid of this problem the object file creation was introduced. This means that single translation units can be handled independently and the binary dump of the intermediate representation can be reloaded in a later step (e.g., linker step).

- Plugin system: The plugin system can be seen as a framework that helps to modify the behaviour of the compiler. The basic idea is to provide fixed positions where the user can modify the standard way that is performed by Insieme. This can for example be the injection of source code or the modification of a statement conversion.

- Elements of the C++ frontend: To support the conversion of all elements of C++ it was needed to introduce some new intermediate representation nodes and add the corresponding visitor methods. Just to name some language elements that where contributed by the author:

  Statement expressions

  Goto and Label statements

  Auto type

  Typeid expressions

  Complex types

  Enum types

1.2 Compiler

A compiler is a software tool that can create an executable computer program out of an input file, that is written in a specific programming language. More general a compiler can be described as a computer program that translates a program that is written in programming language A into a semantically equivalent that is written in programming language B. Normally the input source is written in a high level language (e.g., C++) and the produced output is a machine read- and executable language like Assembler or some specific bytecode.

The execution of the compiler and the translation process from one language into another language respectively is called compilation. The reverse process is called decompilation. The core components and the belonging input and output data can be seen in Figure 1.2. The core components of the compiler

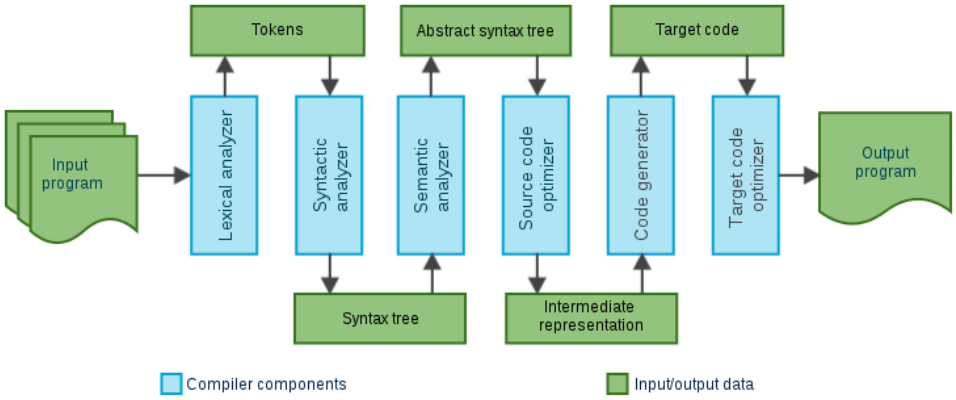

Figure 1.2: Compiler workflow [3]

can be grouped into two different areas. The compiler frontend and the compiler backend. Basically everything that happens before the intermediate code exists belongs to the compiler frontend and all steps that follow the intermediate representation creation belong to the compiler backend area. Section 1.2.1 describes the components of the compiler frontend and backend.

The basic standard steps a compiler is performing during the compilation of a program are [4]:

- Syntax checking: The compiler has to check if the input program is written correctly. This means that the source code has to apply to the language syntax. If errors are found the compiler has to report them to the user.
- Analysis and optimization: The intermediate representation is analyzed and optimized. The power of this feature set is different from compiler to compiler and can also be controlled by the user.
- Code generation: Once the intermediate representation is translated, the compiler tries to create a semantically equivalent program written in the target language. Target language optimizations can be done in this phase.

1.2.1 Compiler infrastructure

Compiler frontend

The main functionality of the compiler frontend is to prepare the input code for parsing, syntactic and semantic analysis and error recognition. These key

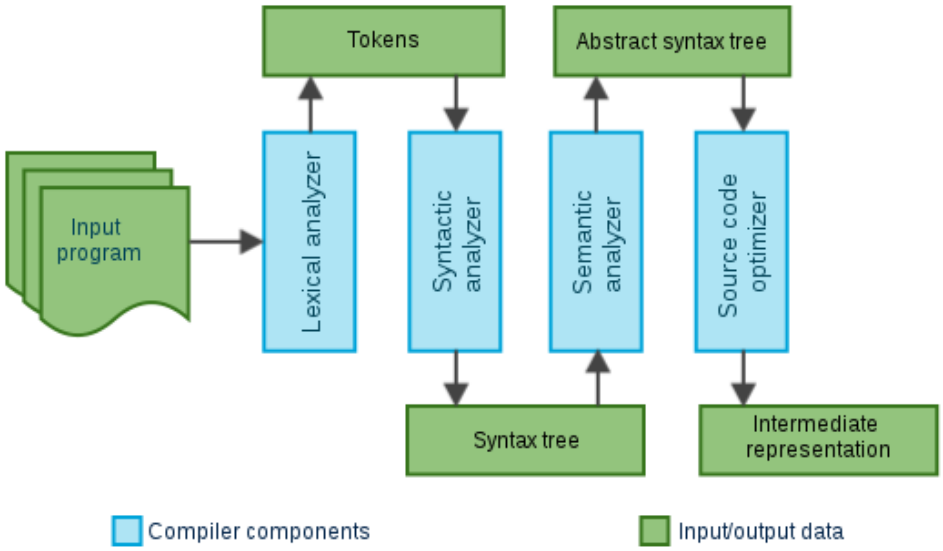

Figure 1.3: Compiler frontend [3]

functionalities are grouped into several phases that can be seen in Figure 1.3.

Compiler frontend components:

- Lexical analyzer: The lexical analyzer component gets the input source code and splits it up into the most basic language elements. The most basic elements are called tokens and can for example be operators, keywords, identifiers, and numbers. Once the lexical analyzer finds no more tokens it returns the list of tokens. If there are tokens that cannot be mapped to a token group the compiler throws a lexical error.

- Syntactical analyzer: The main task of the syntactical analyzer is to check if the token list matches the grammer of the source language. This means for example that after an if keyword there has to be an opening bracket. The output of the syntactical analyzer is a syntax tree.

- Semantic analyzer: The semantic analyzer checks if the given syntax tree applies to the semantic rules of the source language. For example a floating point division can only be applied to floating point data types. The compiler annotates the syntax tree with additional information to recognize mismatches, find all declared variables, etc.

- Source code optimizer: The source code optimizer component is used to perform optimizations on the intermediate representation. Before the source code optimization step, the compiler is generating a more low level language out of the abstract syntax tree. This low level language is called intermediate representation (IR). Such optimizations can for example be constant folding, dead code elimination, and loop fusion.

Compiler backend

All steps that are performed after intermediate representation generation are grouped into the compiler backend. The main functionality of the backend is to use the intermediate language to create correct and semantically equivalent target code. Further features are intermediate and target code optimizations. The basic steps that are performed in the compiler backend can be seen in Figure 1.4.

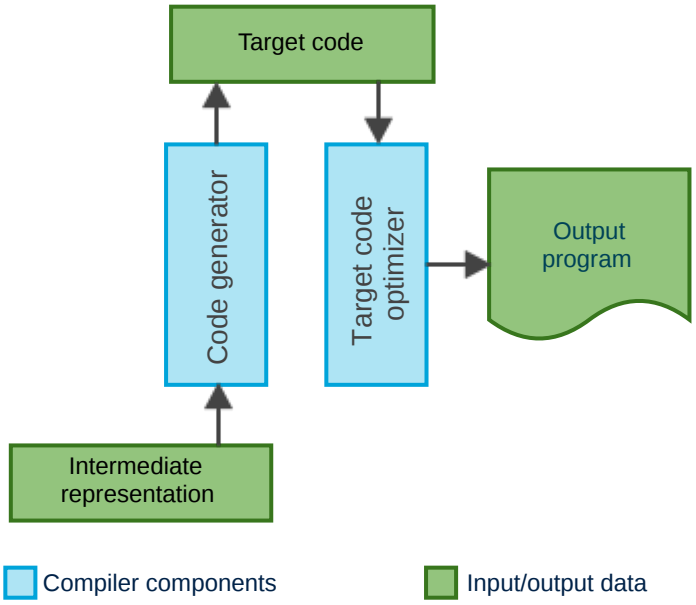

Figure 1.4: Compiler backend [3]

Compiler backend components:

- Code generator: The intermediate representation that was generated before is now converted into the target code. Especially for multi platform

compilers it is important to have an intermediate representation, because this representation can be exchanged between different platforms in order to generate target code for a specific platform. The output that is generated by the code generator can for example be assembler code or a machine code.

- Target code optimizer: The last step is the target code optimization. This optimization step tries to identify slow operations and interchange them with faster operations if possible. This can for example be the replacement of a multiplication by a shift operation.

1.2.2 Classification of compilers

Compilers can be classified based on the way how target code is generated.

- Single pass compiler: A single pass compiler is a compiler that is creating the target code in one single run. This means that all information that is needed can be collected immediately. This is not possible for all languages or language constructs (e.g, C++ forward declarations)

- Multi pass compiler: The target code is not created in a single pass. This means for instance that in a first pass the intermediate representation is generated and in a second step the target code is produced. Multi pass functionality is needed if the compiler has to resolve language constructs (e.g., C++ forward declarations) that make it impossible to create the target code in a single pass.

- Native compiler: A native compiler can only create target code for the platform where the compiler is running.

- Cross compiler: In contrary to the native compiler a cross compiler can also create target code for architectures that are not of the same type where the compiler is executed. This can be helpful if the target platform is unsuitable for larger compilations (e.g., embedded systems).

- Source-to-source compiler: A source to source compiler is not generating executable code for the target platform. It tries to generate a translation of the input code into another language.

- Just-in-time compiler: The just in time compiler is translating the source code into target code only when it is needed. This means that unused code elements are not translated. An advantage of this kind of compiler is that the compiler has runtime informations and generally more knowledge

than a standard compiler, because the executable is already in a running state when the compilation step is performed.

1.3 Clang compiler frontend

The Clang compiler frontend is a part of the LLVM compiler and was developed as an replacement for the GCC compiler frontend. So far the Clang compiler frontend supports following languages:

- C
- C++ (up to C++11 standard)
- Objective-C
- Objective-C++

The compiler frontend is not limited to be used as a part of the LLVM compiler. This makes it easy for other projects to use parts of the Clang frontend in their own projects. [5]

1.3.1 Features

The following information about the Clang features can be read in a more detailed and complete version in [6].

- Fast compile times and low memory usage: Compared to GCC the Clang compiler frontend produces a more compact abstract syntax tree. This results in a faster processing time and lower memory usage.
- Expressive diagnostics: One of the main advantages of the Clang compiler frontend is the user friendly diagnostic system. Warnings and errors are represented in a readable way. Highlighted related information, coloring of the important terms, and exact position of the error makes it easy to find and fix errors in the input code.
- GCC compatibility: GCC is the de facto standard open source compiler that is widely used. There are a lot of GCC extensions that are not defined in the C or C++ standard. In order to compile code that uses this GCC extensions, the Clang compiler frontend has to support this extensions too. Clang is fully GCC compatible which means that all GCC extensions are implemented in Clang.

- Modular library based architecture: Clang is divided into several libraries and tools. This design makes it easy to use for other developers, because the different elements of the frontend are clearly divided and can be accessed by a well defined interface. For instance, if a developer wants to develop a preprocessor tool the Basic and Lexer libraries can be used. Following libraries are available [6]:
  - libsupport: Basic support library, from LLVM.
  - libsystem: System abstraction library, from LLVM.
  - libbasic: Diagnostics, SourceLocations, SourceBuffer abstraction, file system caching for input source files.
  - libast: Provides classes to represent the C AST, the C type system, builtin functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc).
  - liblex: Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion.
  - libparse: Parsing. This library invokes coarse-grained 'Actions' provided by the client (e.g. libsema builds ASTs) but knows nothing about ASTs or other client-specific data structures.
  - libsema: Semantic Analysis. This provides a set of parser actions to build a standardized AST for programs.
  - libcodegen: Lower the AST to LLVM IR for optimization and code generation.
  - librewrite: Editing of text buffers (important for code rewriting transformation, like refactoring).
  - libanalysis: Static analysis support.
  - clang: A driver program, client of the libraries at various levels.

1.3.2 Performance

The following information and graphic elements stem from the Clang features and goals website [6]. A main objective of Clang is performance and to provide a lightweight infrastructure. Profiling of the compiler can be done in a simple way, because of the library based architecture. In order to test the performance of the Clang compiler frontend following test was performed. The input file called Carbon.h is a huge piece of code with 12.3 million lines of code, 10000 function declarations, 2000 structures, 8000 fields, 20000 enumerations and 558 include directives, etc. Figure 1.5 shows the time performance of the Clang compiler

frontend compared to GCC 4.0. The time that Clang needs for the frontend activities is 2.5 times smaller than the time that is needed by GCC 4.0.

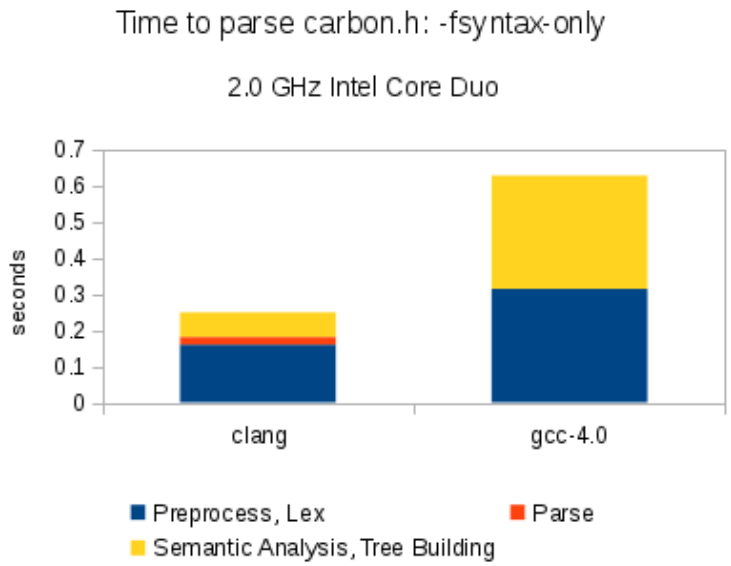

Figure 1.5: Clang time performance [6]

A second important factor is the memory that is needed to store the abstract syntax tree. Again, Clang is much more efficient than GCC 4.0. Figure 1.6 shows the memory usage of the abstract syntax tree for the test input file (Carbon.h). The tree that is produced by GCC is 10 times larger than the preprocessed input file. The tree that is produced by Clang is only 30 percent larger than the preprocessed input source.

1.4 Related work

The first part of this thesis is concerned about the implementation of a widely-usable driver for the Insieme compiler. Because such a driver component is part of every modern compiler, there are many related work topics. For the creation of the Insieme driver the GCC driver was used as a template. The reason for this is that in a standard Linux environment GCC is used as compiler for Insieme generated codes. The main difference between the Insieme driver and the GCC driver is that the Insieme compiler only supports the most important flags (e.g., include-, linker-, optimization-, debug-flags, etc.). Additionally to this, the Insieme compiler driver supports flags to activate or deactivate Insieme internal features like semantic checks, or the recognition of pragmas, etc.

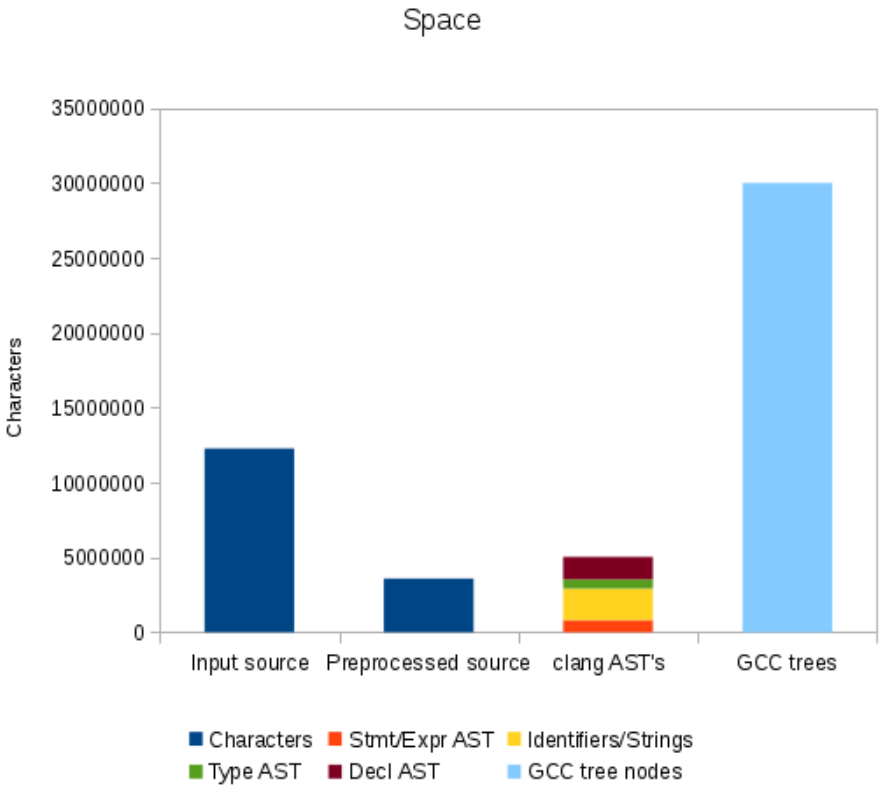

Figure 1.6: Clang space performance [6]

The second main part of the thesis describes the implementation of a plugin system for the Insieme frontend. Other compilers, especially open-source compiler projects, support plugin systems too. The plugin systems of LLVM, GCC, and the Cetus compiler are best comparable to the Insieme frontend plugin system. According to [7] the ROSE compiler infrastructure does not have a documented support for the extension of compiler features with the help of user developed plugins or compiler passes.

- LLVM compiler passes [8]: LLVM supports a wide variety of different compiler passes that can be used to modify the intermediate representation that is generated by the LLVM compiler. It is very easy to modify code constructs like functions, loops, basic blocks, etc. Another use case of the compiler pass system is to provide code analysis features. However, there is no way to modify the behaviour of how the intermediate representation is generated. The passes are not executed during the intermediate

code generation and have to be applied with a tool, that takes the IR code dump and the precompiled compiler pass as input and outputs the modified LLVM IR code. This can be seen as the main difference to the Insieme compiler. The Insieme frontend plugin system does not only provide features to modify the intermediate representation. It is also possible to change the standard way of how Clang AST nodes are converted into IR nodes.

- GCC Plugin API [9]: The GCC plugin API provides means for code analysis, transformations, and optimizations. The plugin callbacks can be invoked at pre-determined events, like it is done in the Insieme frontend plugin system. For instance, the plugin API of GCC supports plugin events after parsing declarations or types, during the garbage collection, before GCC exits, etc. The Insieme frontend plugin system focuses on the compiler frontend and the interfaces to and from other Insieme components and therefore does not support all of the GCC plugin features. Important pre-determined events that are supported by the GCC Plugin API are:
  - after finishing parsing a type
  - after finishing parsing a declaration
  - before a translation unit is converted
  - after finishing a translation unit
  - before GCC exits
  - before GCC Garbage Collection
- Cetus compiler passes [10]: User provided compiler passes can modify the IR that is generated by the Cetus compiler. Like in the LLVM compiler passes there is no way to modify the IR during its generation. According to the documentation an important feature of the Cetus compiler pass system is the iteration through the generated IR. According to [10] there is also a set of different tools that provide a compiler pass template that can for instance be used for:
  - DataFlowTools: Utility methods for detecting used or modified memory accesses.
  - IRTools: Utility methods for searching specific types of IR objects that appear in the IR tree.
  - PrintTools: Utility methods that enable pretty printing of collections and user-directed printing of verbose information.

- SymbolTools: Utility methods related to Cetus' symbol interface.
- Tools: Utility methods for general purpose.
- Expression simplifier: The expression simplifier is used to normalize and simplify IR expressions.

Chapter 2

Insieme compiler project

2.1 Project description

The following description is the mission statement of the Insieme project and can be found in [11]:

> Parallel computing systems have become omnipresent in recent years through the penetration of multi-core processors in all IT markets, ranging from small scale embedded systems to large scale supercomputers. These systems have a profound effect on software development in science as most applications are not designed to exploit multiple cores. The complexity in developing and optimizing parallel programs will rise sharply in the future, as many-core computing systems become highly heterogeneous in nature, integrating general purpose cores with accelerator cores. Modern and in particular future parallel computing systems will be so complex that it appears to be impossible for any human programmer to effectively parallelize and optimize programs across architectures.
>
> 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.
>
> Insieme features:
>
> - 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.
- A rich analysis and transformation toolbox which operates on INSPIRE and aims to maximize developer productivity when researching new optimizations
- Deep integration between the compiler and its associated runtime system, allowing the convenient exchange of arbitrary meta-information for novel combined optimization strategies

2.2 Architecture

The Insieme compiler project consists of two major components. The *Insieme compiler* infrastructure and the *Insieme Runtime System*. The main feature of the Insieme compiler is to translate input code, that is either written in C or C++, and uses OpenMP, MPI, Cilk, or OpenCL, into the Insieme intermediate representation (INSPIRE) [12] and back into C code. The resulting C-representation can be combined with the Insieme Runtime System and can be compiled into an executable application.

2.2.1 Insieme Runtime system

The aim of the *Insieme Runtime System* [13] is to execute parallel programs, that are specified with the INSPIRE language and compiled with the Insieme compiler. This means that a parallel input code (e.g., C++ application that uses OpenMP directives) can be translated into an INSPIRE program that expresses the input source code and additionally the parallel regions, synchronization points, etc. of the code. The code that is generated by the Insieme compiler is making use of the Insieme Runtime System in order to manage the parallel execution. As described in [13], the Insieme Runtime System is able to interact with, monitor, reconfigure the underlying hardware of the system and apply online code-tuning and steering. Data parallelism is supported as well as

task parallelism.

Figure 2.1 shows a basic overview of the elements that form the Insieme Runtime System. It can be seen that the system is split up into three categories [13]:

- Abstraction: The abstraction layer encapsulates system specific implementations of threading, synchronization, affinity settings, etc.
- Core: Components providing the core functionality of the runtime system.
- Utilities: Provides features that are used by the runtime system to accomplish its tasks. This can for example be system specific timers or monitoring functionalities.

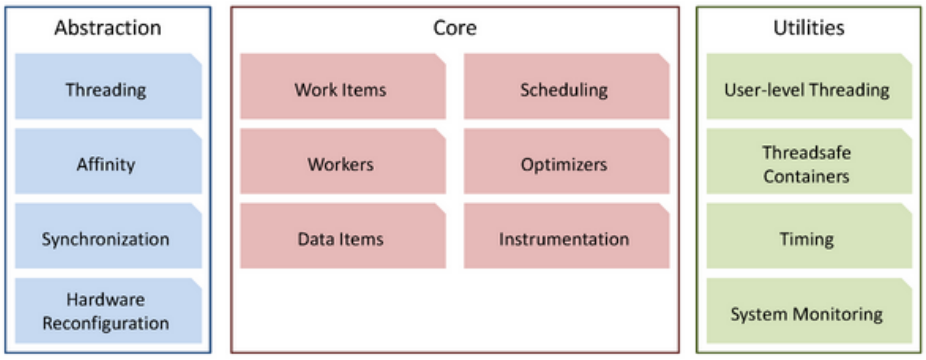

Figure 2.1: Insieme Runtime System [14]

Example: The code blocks that can be seen in Figure 2.2 show an example translation of some input code into the corresponding INSPIRE code and back into C code. The input code is using OpenMP to implement a parallel for loop. The Insieme compiler will recognize this loop and will translate it into a *pfor* (parallel for) INSPIRE element. The last box shows the ouput code that is generated by the Insieme compiler. The needed Insieme Runtime client elements are integrated and the parallel for loop can now be executed with the Insieme Runtime. More details about the Insieme Runtime system model, runtime elements, optimization strategies, etc. can be found in the thesis of Peter Thoman [13].

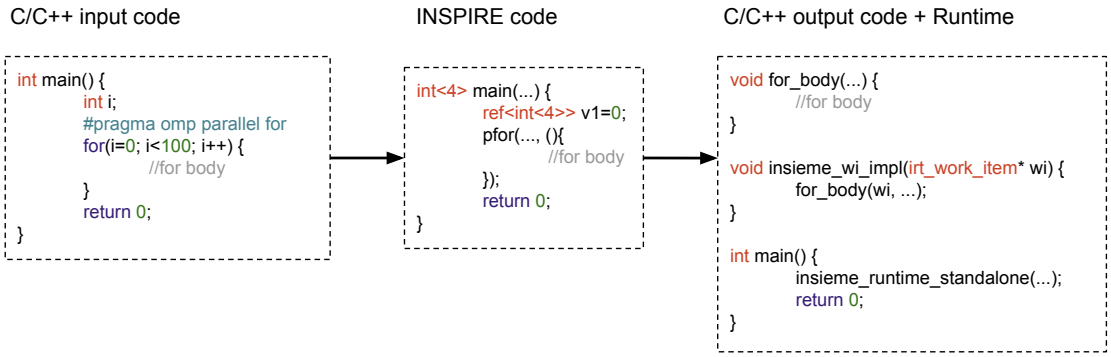

Figure 2.2: Insieme Runtime System example

2.2.2 Insieme Compiler

The second major component of the Insieme project, called the *Insieme compiler*, also consists of several parts. The compiler frontend that is based on different elements of the Clang compiler frontend library, a high level optimizer and a backend component that is producing target code out of INSPIRE code. Figure 2.3 shows the different elements of the Insieme compiler. Further features of the Insieme compiler are different algorithms and techniques for manipulating and optimizing the intermediate representation. The reason why optimization is done in the context of the intermediate representation is because Insieme supports different input languages and parallelization concepts (e.g., OpenMP, OpenCL, and MPI). The generated INSPIRE program represents the parallel concepts in a uniform way, such that one optimizer can be used for different input languages and parallel concepts.

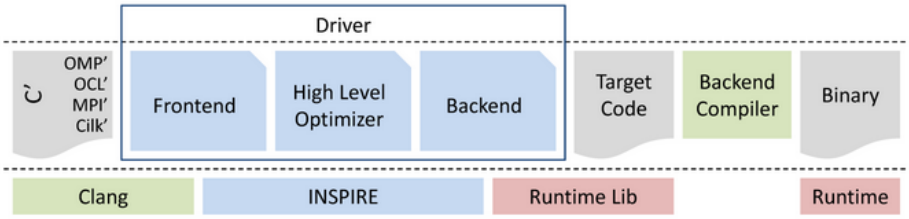

Figure 2.3: Insieme Compiler Infrastructure [14]

Like illustrated in Figure 2.3 the Insieme compiler consists of the following components, among others:

- Driver: The driver component is the entry point of the compiler. The driver can be seen as the interface between end-user and the compiler components. The driver component is setting up the environment, configuring the frontend, core and backend and starts the compilation process. As visible in Figure 2.3 the driver component controls the execution of the frontend, optimizer, and backend.

- Frontend: The main task of the frontend is the translation of the abstract syntax tree (AST), that is provided by the Clang frontend library, into the INSPIRE language. The information provided by parallel programming paradigms (OpenMP, OpenCL, MPI, etc.) is directly included in the intermediate representation.

- Core: The core contains the main functionalities that are needed to create and modify IR language elements. High level intermediate representation analysis, transformations and optimizations are also done in the core.

- Backend: The backend component is used to translate the intermediate representation back into target code. This means that the INSPIRE code is translated into C/C++ code. The runtime library can be used when generating target code in order to create parallel programs that fully utilize the available resources. The target code can be compiled with any C compiler (e.g., GCC or LLVM).

The driver setup, AST creation, and the generation of the intermediate representation is described in detail in the sections of Chapter 3.

Chapter 3

Insieme Frontend

This chapter will explain the steps that are performed to convert input source code to the Insieme intermediate representation. Additionally the entry point of Insieme, that invokes the compiler workflow, and the frontend configuration phase is explained in Section 3.1.

3.1 Driver setup and configuration

The entry point of Insieme is contained in the driver component of the compiler. A driver component can be seen as an interface to the compiler. This means, if the compiler application is started by a user, the code in the main function of the driver will be executed and the actual compiler workflow is started. Insieme has different drivers for the different use cases like analysis, demonstration uses, GCC replacement, etc.

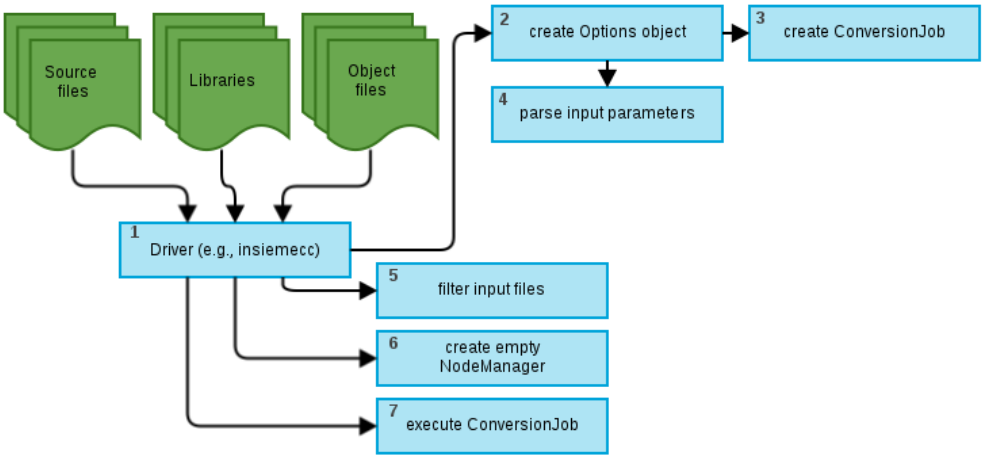

Figure 3.1: Simplified Insieme flowchart

The most useful driver is the *insiemecc* driver that can be seen as a replacement for GCC (e.g., in Makefiles). insiemecc supports most of the important GCC flags and is generating an executable out of the IR code. Basically the driver can be seen as the top level element that is creating the needed objects and invoking the methods that are used to run the core functionalities of the compiler. Figure 3.1 shows the steps of the driver setup and configuration. In the first step the driver is parsing the command line arguments. To accomplish this, the driver is passing the parameter string to the *Option* constructor. The *Option* constructor is generating an empty *ConversionJob* object that will be used to generate the intermediate representation afterwards. After the empty *ConversionJob* is created, the command line arguments are parsed by the *Option* object and the *ConversionJob* is configured. In simple terms the *ConversionJob* object can be seen as a structured pack of information that is extracted from the argument list that is passed to insiemecc.

Example call: *insiemecc testfile.cpp --std=c++11 -O3 -o test.out*

The initial *ConversionJob* has no input files. During the parsing phase the input source file (*testfile.cpp*), optimization flag (*O3*), output file (*test.out*), and C++ standard information (*c++11*) is recognized and the *ConversionJob* is adjusted. List of available parameters (insiemecc, May 2014):

-i [--input-file] arg	input files - required!
-l [--library-file] arg	linker flags - optional
-L [--library-path] arg	library paths - optional
-I [--include-path] arg	include files - optional
-D [--definitions] arg	preprocessor definitions - optional
-f [--fopt] arg	optimization flags - optional
--std arg (=auto)	determines the language standard
--no-omp	disables OpenMP support
--no-cilk	disables cilk support
-o [--output-file] arg	the output file
--intercept arg	regular expressions to be intercepted - optional
--intercept-include arg	intercepted include file - optional
-w [--no-warning]	Inhibit all warning messages
-c [--compile]	compilation only
-S [--strict-semantic]	semantic checks
-O [--full-optimization]	full optimization
--tu-code arg	dump translation unit code
--ir-code arg	dump IR code
--trg-code arg	dump target code

The argument list provides an overview of the diverse configuration options of the *ConversionJob*.

Once the configuration is finished the driver has to decide what input files need to be converted (step 5 in Figure 3.1). Not all input files need to be source files (e.g., one source file and one object file as input). There is the option to pass library files or object files that were converted into intermediate representation and dumped into a binary file in a previous step. Therefore it is necessary to filter out the real source files. All files that are in form of intermediate representation (Insieme objects) or in form of a library should be stored for the backend steps, where the intermediate representation is translated into C or C++ code again. If the input file is neither an Insieme object nor a library it is passed to the target code compiler (e.g., GCC). This filtering procedure can be seen in Figure 3.2.

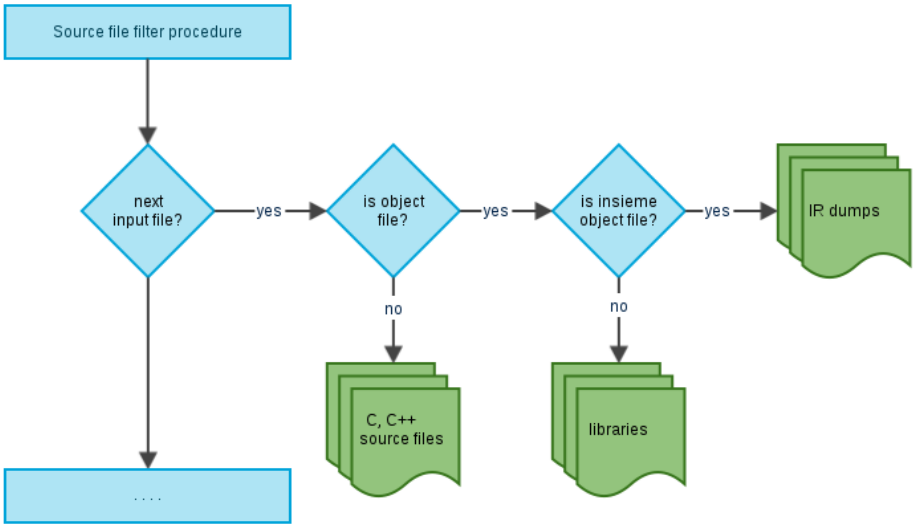

Figure 3.2: Insieme driver input file filter process

In the end insiemecc holds three containers:

- Real source files: C or C++ files that need to be converted into intermediate representation.
- Insieme object files: Files that don't need to be converted again but can be used when creating target code.

- External libraries: Stores the paths to external library files (e.g., libopenmp, libgmp, libboost, ...). External libraries are only used when the target code is compiled to an executable. Therefore the external library information is only needed in the last step.

Once the configuration and file filtering process is done, the real source files can be converted from C or C++ into the Clang AST representation and afterwards into the Insieme intermediate representation.

3.2 Retrieving the Clang AST

Before the actual conversion into the Insieme intermediate representation can start, the input source code needs to be converted into an abstract syntax tree (AST) representation.

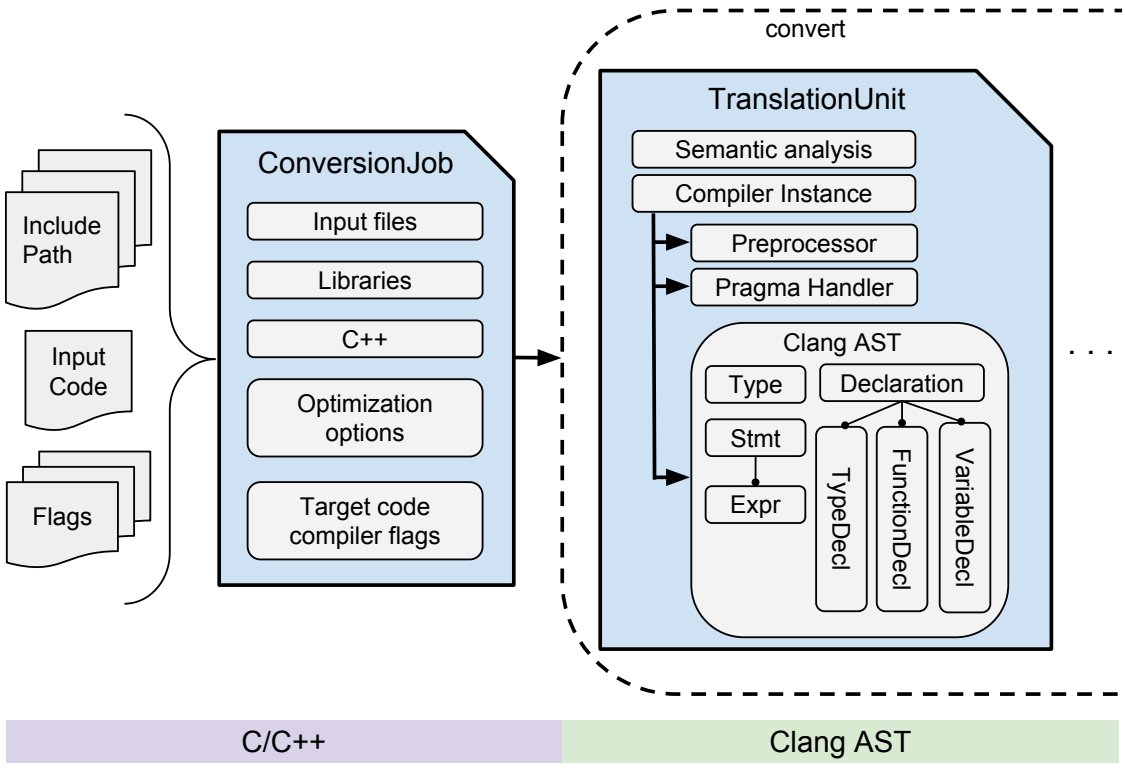

Figure 3.3: Insieme AST creation

```
/** code/frontend/src/frontend.cpp **/

toIRTranslationUnit (NodeManager)
    //initialize plugins
    initalizeFrontendPlugins()
    ...
    //iterate over source files and call convert method
    while hasFile() != NULL
        result = convert(NodeManager, getNextFile(), setup)
    ...
    //post processing
    mergeTranslationUnits(result)
    ...
```

Listing 3.1: ConversionJob execution

Insieme uses the Clang compiler frontend library to create the AST. Figure 3.3 shows the steps that follow the flowchart from Figure 3.1, the corresponding frontend elements, and the most important member fields.

After the initial frontend configuration and setup phase is finished, the *ConversionJob* can be executed. This will initalize the frontend plugins (see Chapter 4) in a first step and call a method that is iterating over the input files. The static *convert* method, that is creating a *TranslationUnit*, is called for every source file. The *convert* method can be seen as a method that takes a *ConversionJob* object and returns an *IRTranslationUnit* object. The steps to generate the Clang AST and create the IR are done in this method. Listing 3.1 shows the pseudocode for the *ConversionJob* execution. Like visible in Figure 3.3, the *TranslationUnit* instance contains a *ClangCompiler* instance and a *InsiemeSema* object. The *ClangCompiler* element can be seen as an interface to the Clang library that is used for the preprocessing, AST generation, and semantic analysis steps. The Clang doxygen describes the *ClangCompiler* as following [15]:

1. It manages the various objects which are necessary to run the compiler, for example the preprocessor, the target information, and the AST context.

2. It provides utility routines for constructing and manipulating the common Clang objects.

The parameters that were collected in the previous steps can now be used to configure the *ClangCompiler*. This can for example be the include directories, preprocessor options that were passed as a command line argument, diagnosis

```
/** test.h **/

int getY() {
  return 200;
}
```

Listing 3.2: test.h input file

```
/** test.c **/

#include "test.h"

#define VALUE 100

int getX() {
  return VALUE;
}

int main() {
  int x = getX();
  int y = getY();

  #ifdef DEBUG
  int z = x * y;
  #endif

  return 0;
}
```

Listing 3.3: test.c input file

```
/** preprocessed code **/

int getY() {
    return 200;
}

int getX() {
    return 100;
}

int main() {
    int x = getX();
    int y = getY();
    return 0;
}
```

Listing 3.4: preprocessed input files

options, etc. After the *ClangCompiler* configuration is completed the pragma handlers are registered. More about pragma handling can be found in Chapter 4.

Before the AST can be generated, the input source needs to be preprocessed. Preprocessing is necessary to provide features like the inclusion of header files, conditional compilation, and macro expansions [16]. The example in Listing 3.2 and 3.3 shows the two input files (*test.c* and *test.h*) before the preprocessing step. Listing 3.4 shows the result of the preprocessed files. It can be seen that all macros are replaced and the include directives are replaced by the contents of the included files.

The next step converts the preprocessed input source code into the Clang AST representation. This step can be seen as a black box, where the input is the preprocessed source code and the result is the abstract syntax tree. The AST

```
/**** AST ****/

TranslationUnitDecl
|-TypedefDecl __int128_t '__int128'
|-TypedefDecl __uint128_t 'unsigned __int128'
|-TypedefDecl __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl <test.h:1:1, line:3:1> getY 'int ()'
| `-CompoundStmt <line:1:12, line:3:1>
|   `-ReturnStmt <line:2:2, col:9>
|     `-IntegerLiteral <col:9> 'int' 200
|-FunctionDecl <test.c:5:1, line:7:1> getX 'int ()'
| `-CompoundStmt <line:5:12, line:7:1>
|   `-ReturnStmt <line:6:2, line:3:15>
|     `-IntegerLiteral <col:15> 'int' 100
`-FunctionDecl <line:9:1, line:18:1> main 'int ()'
  `-CompoundStmt <line:9:12, line:18:1>
    |-DeclStmt <line:10:2, col:16>
    | `-VarDecl <col:2, col:15> x 'int'
    |   `-CallExpr <col:10, col:15> 'int'
    |     `-ImplicitCastExpr <col:10> 'int (*)()'
            <FunctionToPointerDecay>
    |       `-DeclRefExpr <col:10> 'int ()'
              Function 'getX' 'int ()'
    |-DeclStmt <line:11:2, col:16>
    | `-VarDecl <col:2, col:15> y 'int'
    |   `-CallExpr <col:10, col:15> 'int'
    |     `-ImplicitCastExpr <col:10> 'int (*)()'
            <FunctionToPointerDecay>
    |       `-DeclRefExpr <col:10> 'int ()'
              Function 'getY' 'int ()'
    `-ReturnStmt <line:17:2, col:9>
      `-IntegerLiteral <col:9> 'int' 0
```

Listing 3.5: abstract syntax tree

generation is performed in the Clang library. Listing 3.5 shows the AST that is generated out of the preprocessed code from Listing 3.4. It can be seen that the AST contains all the information that is represented in the preprocessed code. For example the three functions *getX()*, *getY()* and *main()* can be seen in line 7, 11 and 15. Before the Clang AST can be used for futher conversion the semantic analysis is performed. If any errors occur, the conversion will be interrupted and the error messages are printed. The pseudocode of the *ClangCompiler* intialization, the pragma handler registration, and the AST creation can be seen in Listing 3.6.

```
1  /** code/frontend/src/translation_unit.cpp **/
2
3  TranslationUnit (NodeManager, Setup, File)
4      //initialize compiler instance
5      mClang = ClangCompiler(Setup, File)
6      ...
7      //register pragma handlers
8      registerPragmaHandler(mClang.getPreprocessor())
9      ...
10     //generate the AST
11     parseClangAST(mClang, mSema, ...)
12     ...
13     //errors occured
14     if errors = true
15         printErrors(mSema)
16         exit;
17     ...
```

Listing 3.6: TranslationUnit listing

3.3 Converting the AST to INSPIRE

The Clang AST can now be converted into the intermediate representation of Insieme (named INSPIRE). This process can be seen as the main and the most complicated part of the Insieme frontend. Figure 3.4 shows the extension of Figure 3.3 and the steps that follow the AST creation, the corresponding frontend elements, and the resulting *IRTranslationUnit*. It can be seen that after the AST creation (first part of the static *convert* method, see Figure 3.3) the second part of the *convert* method is creating an Insieme *Converter* object. The *Converter* is checking if the input file is a C++ or a C file to decide what kind of converter for the type, expression, and statement nodes should be initialized.

Listing 3.7 shows the INSPIRE code that is produced out of the AST from Listing 3.5. As mentioned before, the input code is in the abstract syntax tree representation and the result is in the Insieme intermediate representation. Basically the *Converter* is iterating over all nodes of the input tree and creates an equivalent intermediate representation language element out of it. For example the function declaration *getY* from Listing 3.5 (line 7) is translated into a function called *fun000* that has return type *int*, is taking no arguments, and consists of a *compound* statement that contains a *return* statement.

The last line of the intermediate representation contains the entry point

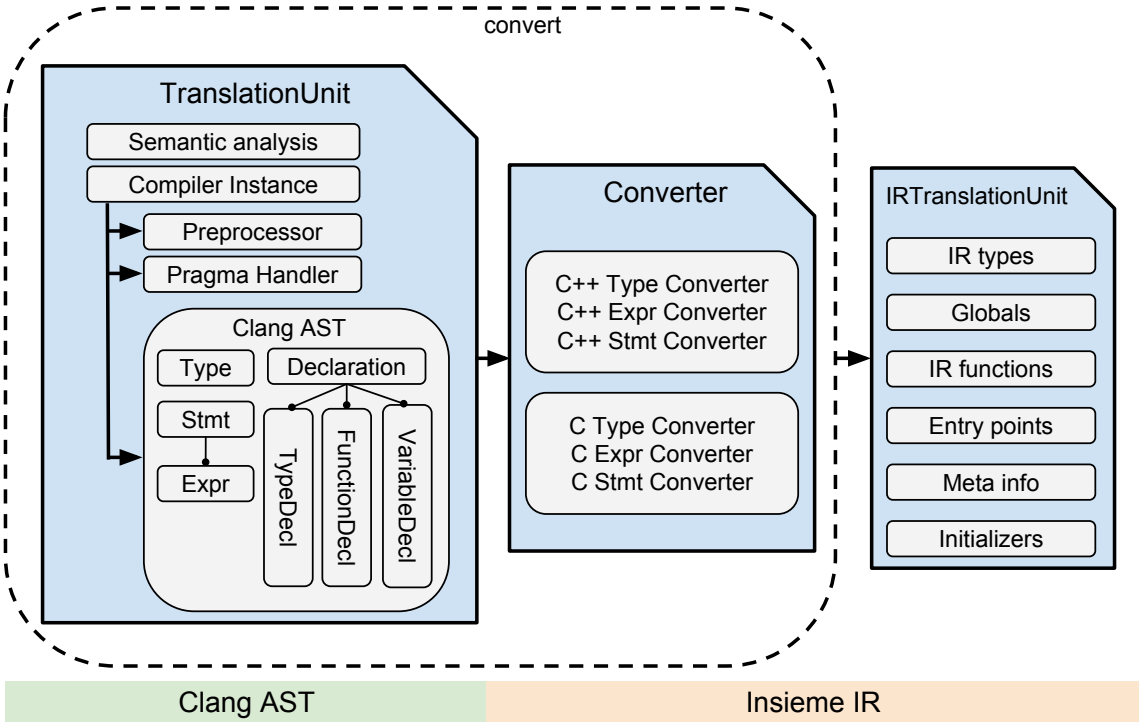

Figure 3.4: Insieme IR creation

information. This information is used to decide what function should be the main function of the backend code. The components that are required to convert AST nodes into intermediate language and the corresponding workflows are explained in Section 3.4 and more detailed in Bernhard Höckners thesis [17]. The result of the conversion into INSPIRE is an *IRTranslationUnit* object that contains the generated IR types, IR functions, information about classes, entry points, etc. (see Figure 3.4).

3.4 Frontend components

The core components of the Insieme frontend can be seen as the most important part. Basically there are four parts that are created and held by the *Converter* object (see Figure 3.5). The top layer element is formed by the declaration visitors that are required to collect information about type declarations, function declarations, and global variable information. The declaration visitors are using the remaining three parts, that form the second layer of the frontend

```
1  //*********************************************//
2  // Pretty Print INSPIRE                        //
3  //*********************************************//
4  let fun000 = fun() -> int<4> {
5      return 100;
6  };
7
8  let fun001 = fun() -> int<4> {
9      return 200;
10 };
11
12 let fun002 = fun() -> int<4> {
13     decl ref<int<4>> v1 =  var(fun000());
14     decl ref<int<4>> v15 =  var(fun001());
15     return 0;
16 };
17
18 // Inspire Program
19 //  Entry Point:
20 fun002
```

Listing 3.7: Insieme IR code

core components. The second layer consists of a type-, expression-, and a statement-converter that is either for C++ or C (see Figure 3.4). Those three elements consist of a collection of methods that are used to convert the different AST nodes into Insieme IR. The generation of IR elements is performed in the Insieme core library. The behaviour of the Clang AST node conversion and the layer structure of the main frontend parts can also be seen in Figure 3.5. For example the *getY* function from Listing 3.4, that can be seen in the AST form in Listing 3.5 (line 7-10), will be recognised first by the function declaration visitor. The function declaration visitor is forwarding the compound statement to the compound statement visitor method of the statement converter. This method is scanning the compound statement and will find the return statement that will be passed to the return statement visitor method. A return statement has a type and an expression and therefore the type converter and the expression converter is invoked. Finally, when all sub-elements are visited and converted, the return and compound statement is converted into IR and the function declaration is translated into the corresponding IR function element.

Declaration visitor elements: The declaration visitor component contains several parts that are used to handle the different types of declarations. This can for example be function declarations, type declarations, etc.

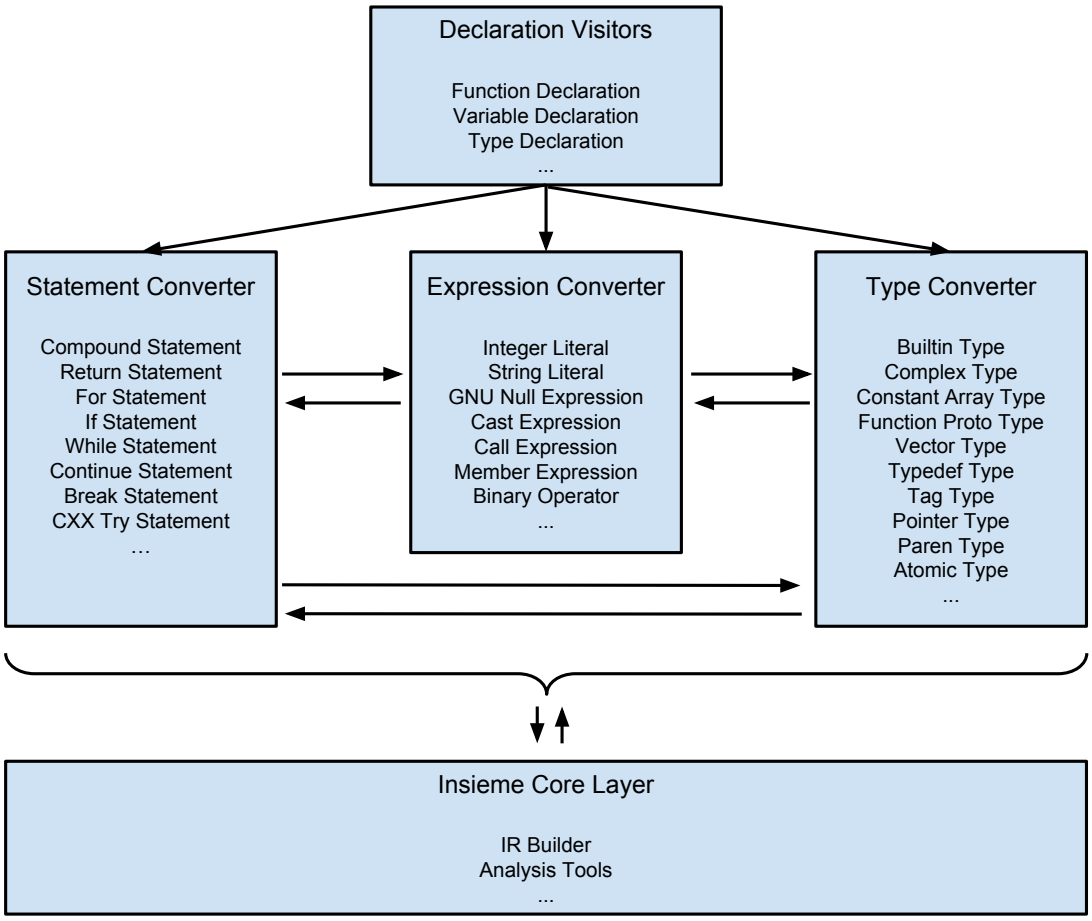

Figure 3.5: Insieme frontend core components

- Type declaration visitor: The type declaration component is used to handle record declarations and typedef declarations. This is done as a first step, to collect the whole information of the types that are used in a single translation unit.

- Global declaration visitor: In order to find all global variables Insieme is making use of a variable declaration visitor that is checking if a variable declaration has a global flag. The global information has to be known for several transformations that are done when performing optimizations, pragma implied modifications, etc.

- Function declaration visitor: Insieme is iterating over all function declarations to convert them into IR functions.

The core components of the Insieme frontend and the detailed description of the declaration visitors, conversion process, and the way it is implemented can be found in Bernhard Höckners master thesis [17].

Chapter 4

The plugin system

In order to provide a high productivity Insieme frontend, it was necessary to implement a plugin system for the Insieme frontend. A plugin infrastructure was already integrated in the core system and the backend of the compiler. The main reasons, to create a plugin system for the Insieme frontend are to:

- provide support for user implemented functionalities
- change the behaviour of the compiler frontend in an easy way
- keep a clean frontend core that supports the standard features and handle everything else with plugins
- help the compiler developers to implement features without touching the frontend core files or adding exceptions to the standard workflow

Example: The C++11 support of the Insieme compiler is done with a frontend plugin. If the standard flag (see Section 3.1) in the compiler call is set to C++11, the plugin is activated and is taking over the conversion of C++11 related Clang AST nodes (e.g., AutoType Clang AST nodes).

Basically the Insieme frontend stores a list of registered plugins. On some well-defined events during the execution of the frontend (e.g., after the conversion of a Clang AST node into IR), the plugins can interact with the compiler. Each of those well-defined events, that will be explained in the sections of this chapter, can be assigned to one of the following plugin phases:

- Clang frontend phase: The actual functionality of the plugin should happen before the AST is generated. This means that the plugin features are executed before the Clang compiler frontend is invoked.
- Conversion phase: The plugin is acting during the conversion of the AST nodes into intermediate representation. The plugin methods are called before the IR is generated.

- Post conversion phase: The Post conversion phase is taking place after the IR generation of an element is completed. For example the plugins are called after an AST node is converted into IR.
- IR phase: The IR phase plugin functionalities are executed after all AST nodes are converted and the IR generation is done.
- Pragma handling: This feature is used to handle pragma directives. It cannot be assigned to a particular compiler phase, because it is acting before the Clang preprocessor is invoked, but can also act after the IR nodes are generated. The first occurrence of the pragma handling functionality is before the preprocessing is done.

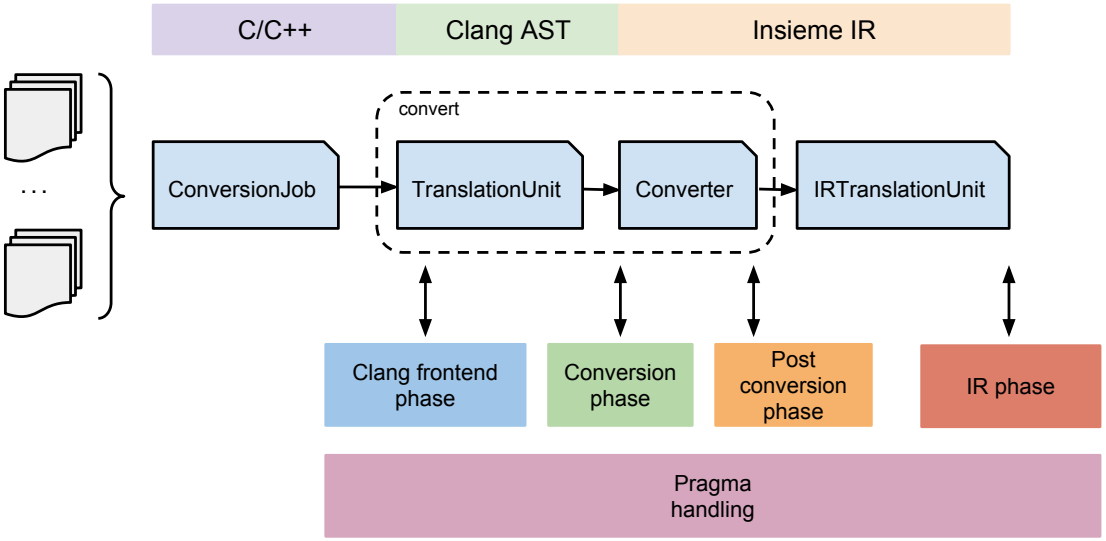

Figure 4.1: Frontend plugin system

In Figure 4.1 the different positions where the plugins can interact with the compiler can be seen. It is also obvious that the plugin calls are done in different compiler phases. For example the Clang frontend phase functionalities of a plugin have to be executed before the AST is generated. As discussed in Chapter 3 and also visible in Figure 4.1, the input parameters are used to create a *ConversionJob* object, a *TranslationUnit* object is invoking the Clang compiler frontend that is generating the AST, and finally the *Converter* is converting the AST nodes into IR and is returning an *IRTranslationUnit* object.

A plugin can act in multiple phases. For example, if a developer wants to create a plugin that is handling user defined pragmas and converting every for loop into a corresponding while-loop, the plugin is acting in several phases.

The following list of frontend plugin examples shows the variability and the diverse application possibilities of the frontend plugin infrastructure:

- OpenMP plugin: Adds support for most of the OpenMP directives. This does not only include the correct recognition of the pragmas. After the recognition, some of the IR statements or declarations (e.g., for statement) have to be modified. In the case of Insieme, the IR nodes are equipped with the uniform parallel constructs that are described in [12].
- OpenCL plugin: Needed to support OpenCL kernels and OpenCL host code.
- Assembler plugin: GCC has some special support for assembler directives that are directly written in the C or C++ source file. The assembler plugin makes it possible to support this GCC proprietary feature in Insieme.
- Semantic check plugin: When compiling bigger projects it gets very difficult to find the origins of mistranslations. The semantic check plugin tries to check every converted node or at least some selected types of nodes for semantic errors. This is a very time consuming plugin but it makes it easier to debug misbehaviour of Insieme.
- Built-in function plugin: GCC provides a large number of built-in functions (e.g., atomic functions). The built-in function plugin adds support for some of those functions.
- Variadic arguments plugin: Frontend plugin that provides support for functions with variadic arguments. This means that the function can have a variable number of arguments.

4.1 The frontend plugin infrastructure

Figure 4.2 shows a simplified overview of how the plugin system is realized. Basically the parent class called *FrontendPlugin*, that is located in the frontend sub-package called *extensions*, can be seen as an interface that contains a set of different methods for the different plugin phases. An arbitrary number of those methods can be overridden by the user created plugins. For example the

TestPlugin2 class, that is illustrated in Figure 4.2 overrides methods for the Clang frontend phase and the IR phase. The reason why there is a need for such an infrastructure, how the system is implemented, and a more detailed description can be found in this section.

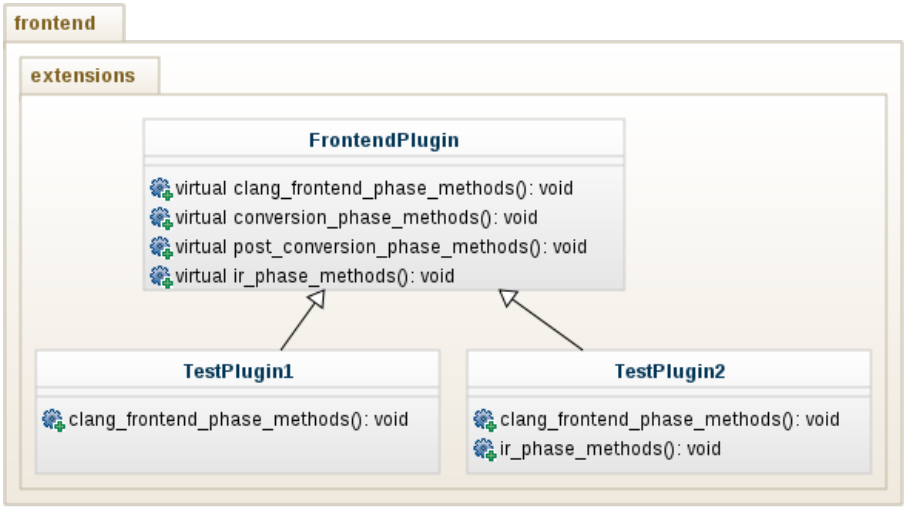

Figure 4.2: Simplified frontend plugin class diagram

One of the main objectives of the frontend plugin system is to provide an infrastructure, that is simple to use on the one hand and usable in many different cases on the other hand. This should be possible without applying any modifications to the core files of the Insieme frontend when implementing a frontend plugin. The main idea is to use fixed positions during the execution, where the frontend is interacting with the plugins. The connection between the plugin system and the Insieme frontend is implemented with the help of an observer pattern [18]. To make the plugin system accessible by different frontend elements (e.g., *Converter*, *TranslationUnit*, etc.) it is required to store the plugin container in an object that is accessible from all needed positions (e.g., after an AST node was converted into IR). The *ConversionJob* element is the best choice, because it is passed to every important frontend element. Figure 4.3 illustrates the connection between the plugin system and the frontend elements.

Example: The *TranslationUnit* object has to access the plugins to execute the methods for the Clang frontend phase. In order to access the registered plugins, the *getPlugins()* method of the *ConversionJob* is called. This method

will return the list of plugins (stored as a member field of *ConversionJob*), that is used to access and call the methods of *ConcretePlugin1* and *ConcretePlugin2*.

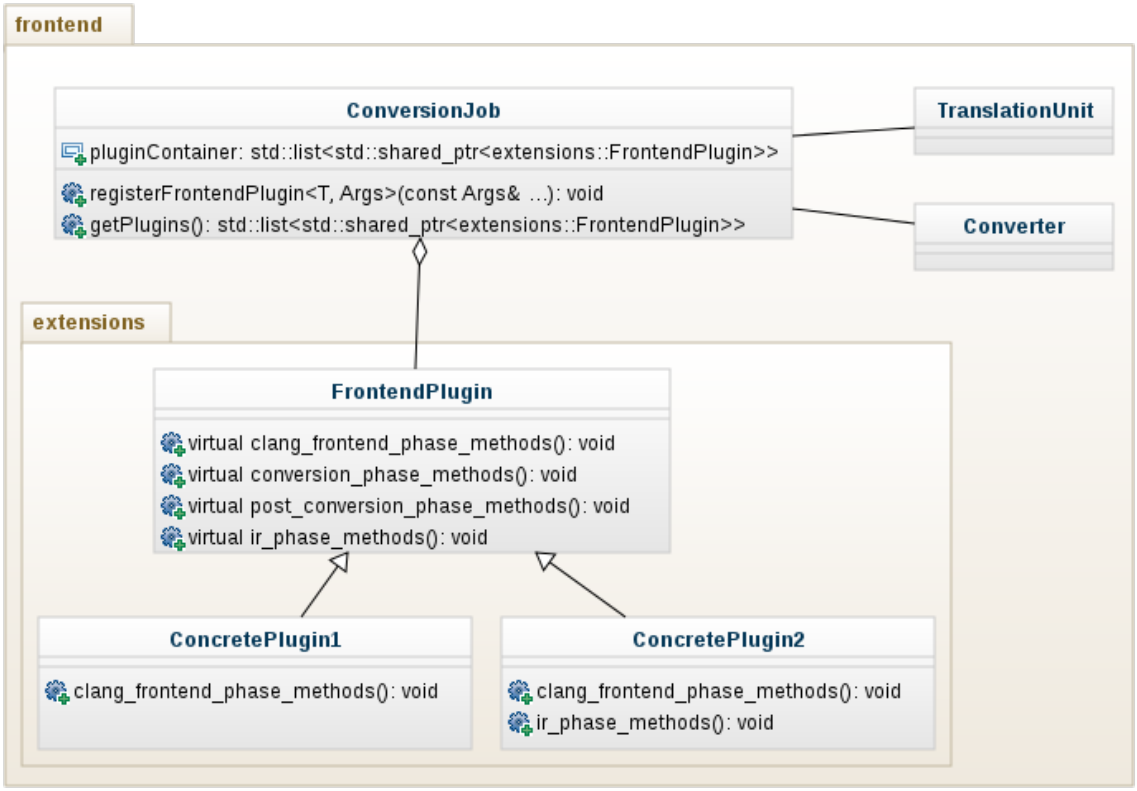

Figure 4.3: Interaction between plugin system and frontend elements

The *ConversionJob* provides a method to register the frontend plugins. At the moment the list of plugins that is used, is hardcoded in the compiler and the registration of the plugins is done in an initialization method of the *ConversionJob*. For future work, we will explore how to make the plugin registration more flexible by using a flag in the *insiemecc* call (see Chapter 5). The plugin container is storing the plugin instances in form of shared pointers to avoid memory leaks.

The plugin interaction positions of the different phases, how they are implemented, and the functionalities of the different plugin phases are described in the Sections 4.2 to 4.6.

4.2 Clang frontend phase

As described in Section 3.1 the Insieme compiler is generating a *ConversionJob* out of the input parameters. The plugins are initialized and registered, and as described in Section 3.2, the *TranslationUnit* instance is created. This instance will create a *CompilerInstance* object that is used to communicate with the Clang library. Before the source code of the actual translation unit can be converted into an AST representation, some options have to be set (see Section 3.2). This is the place where the plugins can interact with Insieme to modify the standard behaviour of the compiler, before the Clang library is used and before the AST is generated.

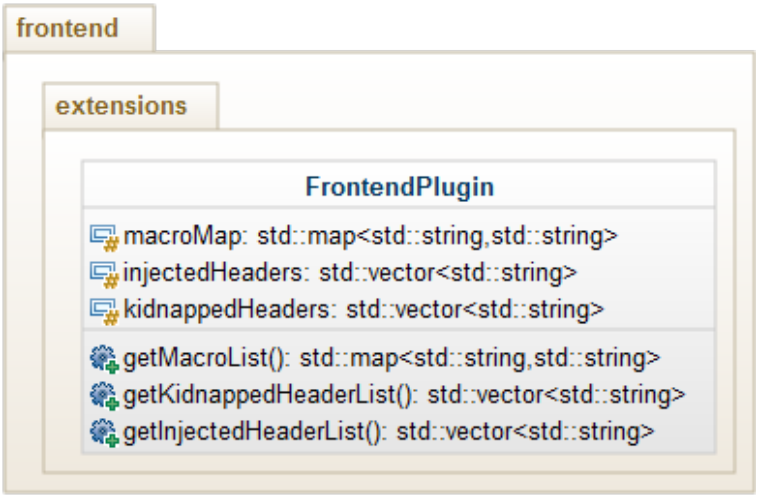

Figure 4.4: Partial class diagram of the FrontendPlugin class (Clang frontend phase)

The partial class diagram of the *FrontendPlugin* class that can be seen in Figure 4.4 shows all methods and data structures that are used in the Clang frontend plugin phase. A full class diagram of the *FrontendPlugin* class is illustrated in Figure 4.12. The Clang frontend phase plugins are supporting the following features. In order to make the description easier to understand this master thesis uses a sample plugin that is shown in Listing 4.1 and a test input source code that is shown in Listing 4.2.

- Inject header files dynamically: This feature can inject header files into different translation units. For example if the user wants to inject function calls that are declared in a non included header file, the plugin can inject the missing header files. The implementation of this feature is quite simple.

```
#include <string>

#include "insieme/frontend/extensions/frontend_plugin.h"

using namespace insieme::frontend;
using namespace string;

class SamplePlugin : extensions::FrontendPlugin {

public:
SamplePlugin() {
    //plugin provided macro definition
    macros.push_back(
        std::make_pair<string, string>("VALUE", "3+5")
    );

    //header injection
    injectedHeaders.push_back("/tmp/header_dir/injected.h");

    //header kidnapping
    kidnappedHeaders.push_back("/tmp/header_dir");
}

};
```

Listing 4.1: Clang frontend phase plugin sample

```
#include <signal.h>

int main() {
  //get the plugin macro def.
  int x = VALUE;

  //if it is 8 raise signal 9
  if(x == 8)
    raise(9);

  return 0;
}
```

Listing 4.2: Clang frontend phase input code sample

The plugin contains a vector (see *injectedHeaders* in Figure 4.4) that stores the paths to the headers that should be injected. During the instantiation of the *CompilerInstance* (as described in Section 3.2) the path vector of each plugin is iterated and added to the list of include files. The output source code of the test input code contains the injected header include directive (visible in line 7 in Listing 4.3).

- Kidnap header files: Sometimes it is required (especially in the Insieme environment) that header files get replaced by other implementations. This can for example be some special implementation of OpenMP calls. The implementation of this feature is similar to the inject header files functionality. The only difference is that the plugins are containing a vector that stores paths to folders. This folders contain the header files that should be exchanged. During the instantiation of the *CompilerInstance* the header search path is configured and all plugin provided paths are added before the system provided ones. When the Clang compiler frontend is looking for a header file it will find the plugin provided path before the system provided path and will use the plugin provided header file instead of the system provided. The test input code from Listing 4.2 is including a file called *signal.h*. If the sample plugin is activated and the developer provides a replacement for the *signal.h* file (by creating a file called *signal.h* in `/tmp/header_dir` and implementing the called functions) a different implementation of the raise method can be used for example.

- Add special macro definitions: Frontend plugins can contain a list of pairs. The pairs consist of a macro name and a definition. Again the *CompilerInstance* is taking all plugin provided macros and is adding the information to the Clang compiler frontend macro list. If the sample application is compiled without registering the sample plugin before, the compilation will fail, because the macro VALUE is unknown. If the code is compiled with the activated sample plugin the generated code contains the substituted macro that can be seen in line 12 in Listing 4.3:

4.3 Conversion phase

The Insieme frontend is interacting with the plugin system after the AST generation. Because the AST generation is performed in the Clang library, it is not possible to modify the AST during its creation. The Conversion phase methods offer the functionality to take over the conversion process of AST nodes. This means, that instead of using the Insieme frontend standard conversion process

```
/**
 * ------------- Auto-generated Code -------------
 * This code was generated by the Insieme Compiler
 * -----------------------------------------------
 */
#include <signal.h>
#include <injected.h>
#include <stdint.h>

/* ------- Function Definitions --------- */
int32_t main() {
  int x = 3+5;
  if(x == 8)
    raise(9);
  return 0;
}
```

Listing 4.3: Clang frontend phase output code

to create IR nodes out of AST nodes, it is possible to use a plugin. Figure 4.5 shows the workflow of an example conversion of an AST node into IR. It can be seen that the AST node (that can be a declaration, a statement, a type or an expression) can be converted either by the Insieme frontend or by one of the registered plugins. This can for example be useful if a compiler developer wants to implement the conversion of non standard AST nodes. One example for this would be the conversion of assembler statement nodes, because they can have different shapes, according to which operating system is used. For instance, *GCCAsmStmt* for code that should be compiled with GCC or *MSAsmStmt* nodes for code that should be compiled with a Microsoft compiler. Figure 4.6 shows all methods that can be used to implement the Conversion phase plugin features. A full class diagram can be seen in Figure 4.12. The Conversion phase plugins are supporting the following features. To make the descriptions of the Conversion phase features easier to understand, this master thesis uses a sample Conversion phase plugin (see Listing 4.4) and a sample input code (see Listing 4.5).

- Plugin based type conversion: Figure 4.5 shows that either one of the plugins can convert an AST type node or the standard path is taken. The standard path uses the Insieme frontend type declaration visitor and type converter. Sometimes it is necessary to avoid the standard path, because it might be the case that the standard type converter has no implementation to convert the given AST type node. This is for example the case when using the C++11 *auto* type. The only way to produce IR code out of an *auto* type AST node is to use the C++11 plugin that

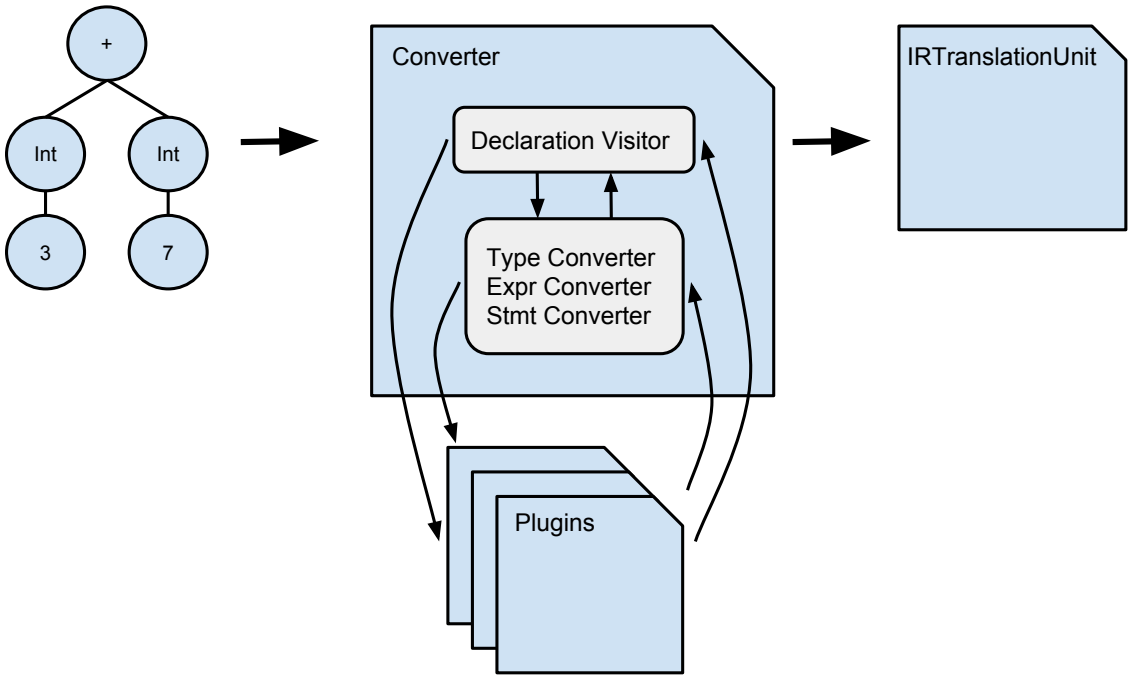

Figure 4.5: AST node conversion example

supports the conversion of *auto* types. The sample plugin from Listing 4.4 is for example looking for floating point types (see line 15-22). If a floating point type is found, it will be exchanged by a long double type. If we use the input code from Listing 4.5 and compile it with the registered sample plugin the *float* from line 5 will be exchanged by a *long double* (see Listing 4.6 line 10).

- Plugin based expression- and statement conversion: The expression- and statement-conversion can also be handled by a plugin. As mentioned before, this can be helpful when trying to implement non standard AST nodes that should be supported by Insieme. An example for such a node type would be the *AtomicExpr* nodes. Due to the fact that atomic expressions are compiler built-ins, it was required to implement this behaviour in Insieme. Another example would be the *MSDependentStmt*. This statement takes a literal as argument and checks if this literal is known at the statement position. The input test code from Listing 4.5 is checking if literal z is known at line 8. This will evaluate to false and the sub statement will not be executed. The sample plugin (see Listing 4.4 line 35-51) implements a method that is looking for those *MSDependentStmt*

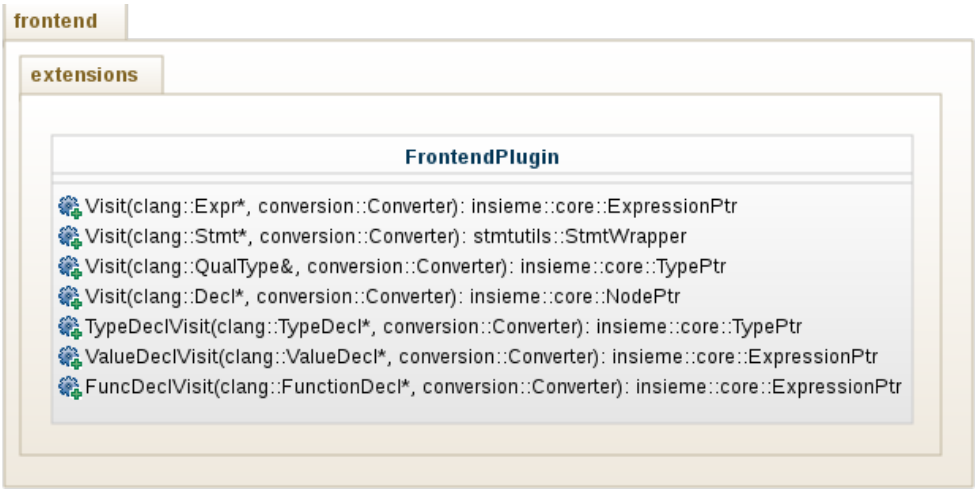

Figure 4.6: Partial class diagram of the FrontendPlugin class (Conversion phase)

nodes. Once such a node is found, the plugin checks if it is an *__if_exists* or *__if_not_exists* statement. The second check examines whether the literal exists. An XOR operation of the two booleans decides if the statement will be converted or simply dropped. In the case of the input test code the plugin will recognize that the literal does not exist and skip the conversion of this node, because it will never be executed. The output code doesn't contain the statement and the corresponding sub-statement anymore. If the plugin returns *nullptr* to the Insieme frontend the standard way will be taken. This can be seen as a marker that tells the Insieme frontend that the plugin does not want to do anything with the current node.

- Plugin based declaration conversion: It is also possible to handle the conversion of a whole declaration. The kind of the declaration can be a *TypeDecl* (covers type information, template information, record declarations, typedefs, etc.), *FuncDecl* (covers functions, constructors, destructors, etc.), or a *ValueDecl* (covers variable declarations, enum constant information, etc.). The prototypes of the virtual methods can be seen in the partial class diagram in Figure 4.6.

Plugins can interfere with the Insieme compiler (during the Conversion phase) before a declaration is visited. As described in Section 3.4 the declaration visitors are using type-, expression-, and statement-converters to convert the Clang AST nodes into IR elements. Those converters are implemented in form of a visitor pattern [19], and therefore there is a method that is delegating the current AST node to the correct converter method. This top level method will

```
#include <string>

#include "insieme/frontend/extensions/frontend_plugin.h"

using namespace insieme::frontend;
using namespace insieme::core;
using namespace std;

class SamplePlugin : public extensions::FrontendPlugin {

public:
SamplePlugin() { }

//sample plugin type visitor
virtual TypePtr Visit(const clang::QualType& qt,
                      conversion::Converter  conv) {
 if(qt.getTypePtr()->isFloatingType()) {
    IRBuilder ir = conv.getIRBuilder();
    return ir.getLangBasic().getLongDouble();
 }
 return nullptr;
}

//sample plugin expression visitor
virtual ExpressionPtr Visit(const clang::Expr*    ex,
                            conversion::Converter conv) {
 if(const clang::AtomicExpr* atom =
    llvm::dyn_cast<const clang::AtomicExpr>(ex)) {
        //atomic expression IR generation
 }
 return nullptr;
}

//sample plugin statement visitor
virtual StatementPtr Visit(const clang::Stmt*    st,
                           conversion::Converter conv) {
 if(const clang::MSDependentExistsStmt* ms =
   llvm::dyn_cast<const clang::MSDependentExistsStmt>(st)) {
        //check if it is an __if_exists statement
        bool isIfExists = ms->isIfExists();
        //test if literal exists
        bool exists = false;
        ...
        if(!(isIfExists ^ exists)) {
            //convert sub statement and return it
            conv.convertStmt(ms->getSubStmt());
        }
        return nullptr;
 }
 return nullptr;
}
};
```

Listing 4.4: Conversion phase plugin sample

```
#include <iostream>

int main() {
    //type visitor test
    float x = 0;

    //stmt visitor test
    __if_exists(z) {
        std::cout << "Literal x exists..." <<std::endl;
    }

    //expr visitor test
    int z =  __atomic_load_n(&x, __ATOMIC_RELAXED);
    return 0;
}
```

Listing 4.5: Conversion phase input code sample

check if a plugin converts the current AST node before the standard conversion method is invoked. In summary it can be said that a plugin can act in the Conversion phase before a declaration is visited and before a type-, expression-, or statement-conversion.

4.4 Post conversion phase

Sometimes the compiler should only modify some small IR code regions (e.g., add a const flag to a C++ reference). Therefore it is convenient to have an interception position after an AST node was converted into IR. The plugin system provides such a feature. Basically everything that can be done in the Conversion phase can also be done in the Post conversion phase. The only difference is that the Insieme standard path was already taken when the plugin system gets the chance to modify the IR. The Conversion phase features should be used when an AST node conversion should be done by a plugin. The Post conversion phase features should be used when the result of the standard conversion should be modified. Figure 4.7 shows a partial class diagram of the *FrontendPlugin* class that illustrates the methods that can be used for IR modifications. The full class diagram can be seen in Figure 4.12.

The Post conversion phase plugins are supporting the following features:

- Modification of generated IR nodes: The plugin is able to change the result of an conversion from AST to IR. This means that the plugin can change the result of the conversion without modifying the input AST or changing the standard way of the Insieme frontend. The AST information

```
/**
 * ------------- Auto-generated Code -------------
 * This code was generated by the Insieme Compiler
 * -----------------------------------------------
 */
#include <stdint.h>

/* ------- Function Definitions --------- */
int32_t main() {
    long double x = (long double)0.0;

    //atomic expression
    ...
    return 0;
}
```

Listing 4.6: Conversion phase output code

is still available and can be read by the plugin. This can for example be useful when a plugin wants to check if the conversion was performed in the correct way. The Clang AST node is used to check if the generated IR code contains the same information. Currently the Post conversion phase provides visitors for the following node types:

Expressions

Types

Statements

Function declarations

Value declarations

Type declarations

Because the Post conversion phase plugin functionalities are very similar to the Conversion phase features, this document does not provide examples for all of the Post conversion phase features. To give a basic overview of how to use the Post conversion phase methods, this master thesis uses a sample plugin that is shown in Listing 4.7. The plugin is using the *PostVisit* method for IR expressions. This means that the plugin is called after each conversion that produces an IR expression node. The plugin gets the original AST node, the converted IR node, and the *Converter*. In the sample plugin the original source location is extracted from the Clang AST node. The Clang *SourceManager* that can be gathered from the *Converter* object (see line 19) is used to get the source location. The next step is to extract the filename, the line number, and

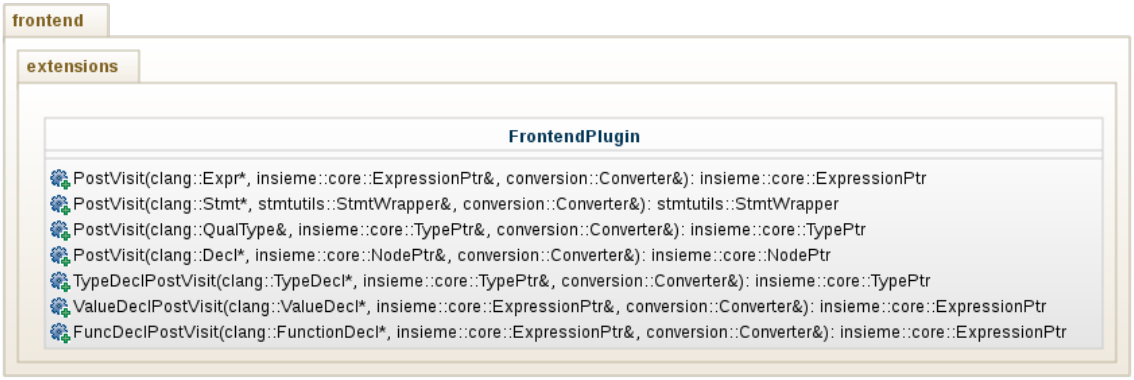

Figure 4.7: Partial class diagram of the FrontendPlugin class (Post conversion phase)

the column number (see line 22-25). Once this is done, the information can be attached to the IR expression node. The original source location can be very useful for debugging issues. Especially if an error occurs after the frontend phase of the compiler.

4.5 IR phase

In order to understand the capabilities of the IR phase plugins it is important to know what happens with the *IRTranslationUnit* object, that is generated for every single input file. Figure 4.8 shows an Insieme workflow example for two input files and the IR phase intervention positions. For each of the two input files a *TranslationUnit*, *Converter*, and finally *IRTranslationUnit* object is created. This is suitable for the creation of IR code, but if Insieme should produce target source code out of the IR code, a single object is required that contains the whole information of all input files (called *IRProgram*). To create an *IRProgram* all previously created *IRTranslationUnit* objects are merged into a single *IRTranslationUnit*, entry points are resolved, type information is merged, etc. The first possibility where the plugin system can change the IR code, or basically the *IRTranslationUnit*, is after all AST nodes of a translation unit were converted (conversion into IR is completed for one translation unit). The second intervention position is after all *IRTranslationUnit* objects were merged into a single unit and an *IRProgram* was created. Information about the merging procedure of *IRTranslationUnit* objects and why this step is necessary can be found in the thesis of Bernhard Höckner [17].

```
#include <string>

#include "insieme/frontend/extensions/frontend_plugin.h"

using namespace insieme::frontend;
using namespace insieme::core;
using namespace std;

class SamplePlugin : public extensions::FrontendPlugin {

public:
SamplePlugin() { }

//sample post plugin type visitor
virtual ExpressionPtr PostVisit(const clang::Expr& ex,
                          ExpressionPtr& irExpr,
                          conversion::Converter& conv) {
 //get the clang source manager
 clang::SourceManager& sm = conv.getSourceManager();

 //extract original source location
 clang::SourceLocation exLoc = expr.getExprLoc();
 std::string file = sm.getFileName(exLoc);
 unsigned line = sm.getSpellingLineNumber(exLoc);
 unsigned column = sm.getSpellingColumnNumber(exLoc);

 //attach location to IR expr node
 annotations::attachLocation(irExpr, file, line, column);
 return irExpr;
}

};
```

Listing 4.7: Post conversion phase plugin sample

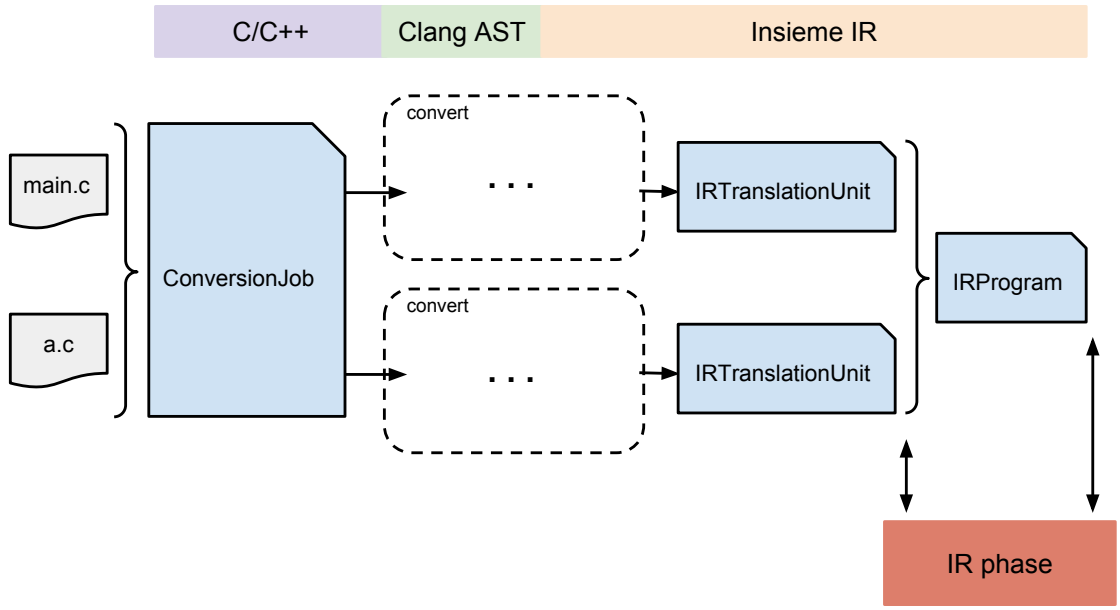

Figure 4.8: Simplified frontend workflow for two input files

The partial class diagram of the *FrontendPlugin* class can be seen in Figure 4.9. As mentioned before, there are two different methods that can be overloaded. One will be called after each *IRTranslationUnit* is completed and one will be called after the *IRProgram* is generated. See Figure 4.12 for a complete class diagram.

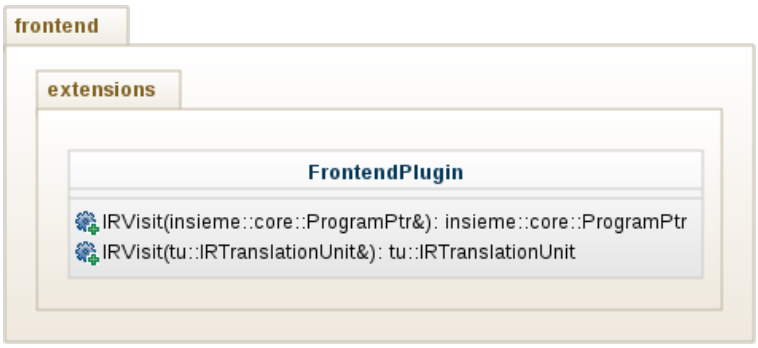

Figure 4.9: Partial class diagram of the FrontendPlugin class (IR phase)

```
#include <omp.h>

int main() {
    #pragma omp parallel for
    for(int i=0; i<100; i++) {
        ...
    }
    return 0;
}
```

Listing 4.8: Pragma handling input code sample

The IR phase supports the following functionalities:

- Modification of *IRTranslationUnit* objects: This functionality provides a method that can modify the contents of an *IRTranslationUnit*. This can for example be used to apply various cleanup operations (e.g., fix wrong return types that are induced when using C++ references). The call to the plugins occurs directly after the static *convert* method returns the *IRTranslationUnit* to the *ConversionJob*, but before the next translation unit is converted. The *ConversionJob*, that calls the *convert* method, calls the *getPlugins()* method to get the list of plugins and iterates through that list to call the *IRVisit* method of each plugin.

- Modification of the *IRProgram*: This feature has the same behaviour as the *IRTranslationUnit* feature. The only difference is that all *IRTranslationUnit* objects have been merged to one single unit before. The call to the plugins is done in the *ConversionJob* object, after all translation units have been converted and merged together. This feature can for example be used to apply optimization operations like unnecessary cast, and superfluous code removal.

4.6 Pragma handling

The pragma handling functionality is a very special feature that cannot be assigned to a particular plugin phase. This section will show that the pragma handling appears in several frontend phases. In order to explain the standard way of how pragmas are handled in Insieme, this master thesis uses a sample input code that can be seen in Listing 4.8.

As illustrated in Figure 4.10 the pragma is attached to a for statement. Due to the reason that a pragma is a preprocessor directive it will first be recognized by the preprocessor of Clang. Insieme holds a map that contains statements

and declarations and the corresponding list of pragmas that is attached to the Clang AST element (either a declaration or a statement). When using the example code the map will contain one *clang::ForStmt* that has one pragma (#pragma omp parallel for) attached. The pragma can be seen in line 4 in the input code. The next step is to create IR code out of the Clang AST elements. During the conversion Insieme will recognize that there is a pragma attached to a Clang node and will therefore create a special IR node (marked node) that contains additional information about the attached pragmas. In the example workflow in Figure 4.10 the code in the orange box illustrates an IR for statement node that is marked with an *OMPAnnotation*. The final IR code can now be passed to a semantic engine that is iterating through all IR nodes and checking if they are marked with an annotation. If a marked node is found, the special handling that implements the needed functionality of the pragma will be applied to the marked node. In the code example the *OMPSema* will recognize the marked node and will add the needed fork and join calls to make the for loop parallel.

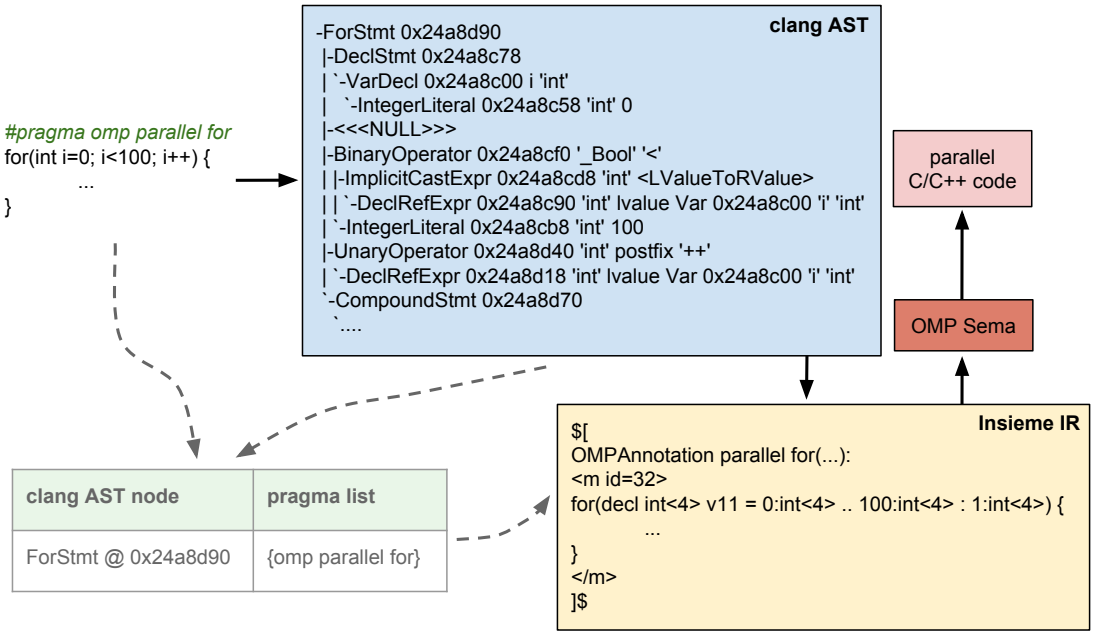

Figure 4.10: Insieme pragma handling example

The pragma handling functionality appears in the preprocessor phase, IR generation and after the IR was generated. More information about how

pragmas are attached to Clang nodes, defining pragma tokens, and how the pragma handling works can be found in the work of Simone Pellegrini [20]. One goal of the plugin system framework is to make the pragma handling more dynamic. This means that it should be possible to implement support for user defined pragmas without changing the Insieme frontend source code. Section 4.6.1 explains the changes that were needed in order to support the recognition of user defined pragma directives. The IR handling for user defined pragmas is described in Section 4.6.2.

4.6.1 User-defined pragma recognition

The class diagram in Figure 4.11 shows all elements that are used to support the plugin based handling of user defined pragmas. It can be seen that the *FrontendPlugin* class holds a container of *PragmaHandler* elements that can be accessed with a method called *getPragmaHandlers()*. The *PragmaHandler* object can be seen as an element that contains the information about a pragma that should be matched (e.g. #pragma omp parallel for) and a function that takes an IR statement that can be modified and returned.

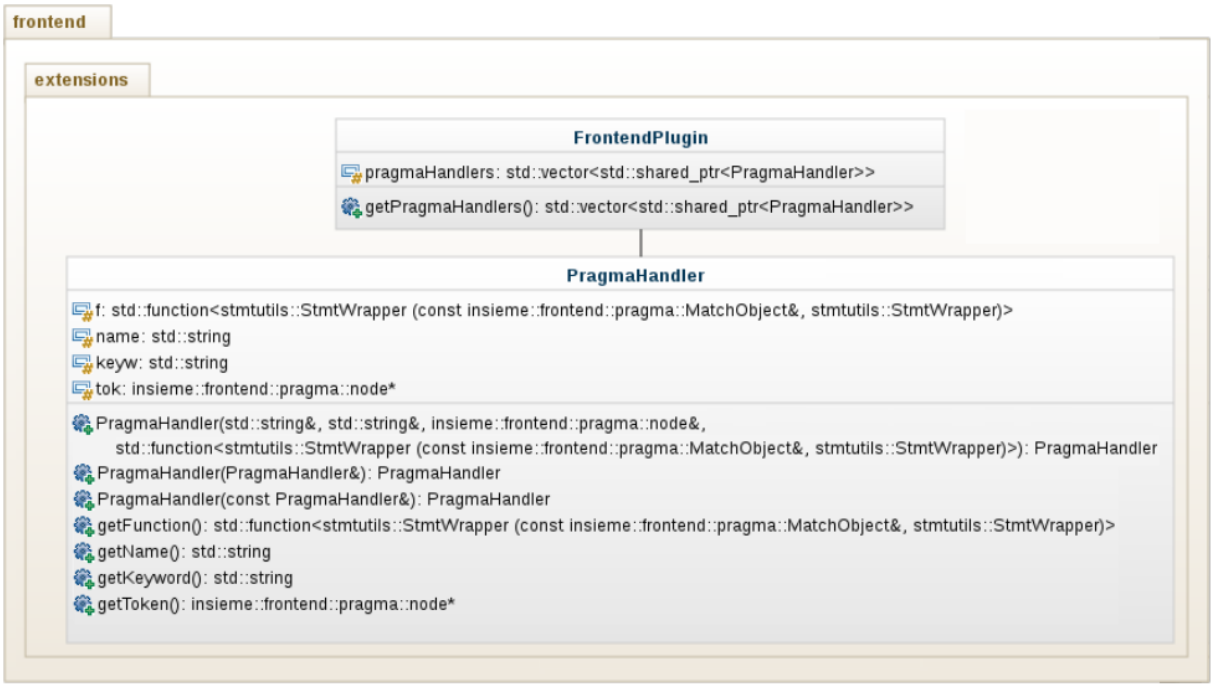

Figure 4.11: Insieme pragma handling class diagram

To support the recognition of user defined pragmas, the Insieme compiler has to register the pragmas that should be matched, before the preprocessing phase and before the Clang AST is generated. Line 8 in Listing 3.6 shows such a registration call. The code that was added to the *TranslationUnit* constructor iterates through the registered frontend plugins and checks if one of the plugins contains a pragma handler. If a *PragmaHandler* is found it will be registered at the Clang preprocessor.

Once the plugin defined *PragmaHandlers* are registered, they will be recognized by the Clang preprocessor and the table that maps pragmas to Clang AST declarations or statements can be built (see table in Figure 4.10). All frontend changes that were needed to recognize user defined pragmas can be seen in Listing 4.9.

4.6.2 User-defined pragma handling

After the Clang AST is generated all nodes will be converted into Insieme IR nodes (see Section 3.3). Due to the reason that a pragma directive can be attached to either a declaration or a statement, Insieme will check after each conversion of a Clang statement or declaration node if this node is contained in the table that maps pragmas to AST nodes. If the node is not found, the converted statement or declaration does not need to be modified. If the AST node is found in the table Insieme will use the list of attached pragmas and call the functions of the *PragmaHandlers* that are responsible for the attached pragmas. The function that is stored in a *PragmaHandler* takes an Insieme IR statement and returns a modified version of the statement.

It may happen that a pragma construct contains several variables or expressions (e.g., #pragma omp parallel for private(a)). The IR nodes of the used elements need to be accessible by the *PragmaHandler* function. Therefore it was necessary to implement a *MatchObject*. The *MatchObject* contains all elements that are used inside of a pragma. For example the *MatchObject* that is passed to the function that handles the pragma directive #pragma omp parallel for private(a) will contain an Insieme IR element for the variable *a* that is used in the private clause.

4.6.3 Simple user-defined pragma example

To explain the workflow of a plugin based user defined pragma handler this master thesis uses the test code shown in Listing 4.10 and a sample frontend plugin that can be seen in Listing 4.12. The first pragma (#pragma test remove) should simply remove the statement that is attached to the pragma.

```
/** code/frontend/src/translation_unit.cpp **/

TranslationUnit (NodeManager, Setup, File)
 ...
 // check for frontend plugins pragma handlers
 // and add user provided pragmas to be handled
 // by insieme
 std::map<std::string,clang::PragmaNamespace *> pragmaNames;
 auto PP = mClang.getPreprocessor();
 for(auto plugin : Setup.getPlugins()) {
  for(auto ph : plugin->getPragmaHandlers()) {
    std::string name = ph->getName();

    // if the pragma namespace is not registered
    // already create and register it and store
    // it in the map of pragma namespaces
    if(pragmaNames.find(name) == pragmaNames.end()) {
      pragmaNames[name] = new clang::PragmaNamespace(name);
      PP.AddPragmaHandler(pragmaNames[name]);
    }

    // add the user provided pragma handler
    pragmaNamespaces[name]->AddPragma(
      pragma::PragmaHandlerFactory::
      CreatePragmaHandler<pragma::Pragma>(
        PP.getIdentifierInfo(ph->getKeyword()),
        *ph->getToken(), ph->getName(),
        ph->getFunction())
      );
  }
 }

 ...
 //generate the AST
 parseClangAST(mClang, mSema, ...)
 ...
```

Listing 4.9: PragmaHandler registration

```
1  #include <omp.h>
2
3  int main() {
4      #pragma test remove
5      if(0==0) {
6          ...
7      }
8
9      #pragma test change return(-1)
10     return 0;
11 }
```

Listing 4.10: Pragma handling input code sample

The second pragma (#pragma test change return(-1)) should change the return value of the following return statement. The pragmas can be seen in line 4 and 9 of Listing 4.10. The sample plugin is defining two *PragmaHandler* objects and is storing them in the pragmaHandlers vector (see line 14-42 in Listing 4.12). As explained in Section 4.6.1, Insieme is registering all pragmas that should be recognized by the preprocessor. In this example Insieme will get two *PragmaHandler* objects that are registered. After the AST is generated the table that maps pragmas to AST nodes contains the contents that can be seen in Table 4.1. After the conversion of the *clang::IfStmt* into the

AST node	**pragma**
clang::IfStmt @ <MemoryAddress>	#pragma test remove
clang::ReturnStmt @ <MemoryAddress>	#pragma test change return(-1)

Table 4.1: Pragma map contents

corresponding IR statement, the function that is attached to the #pragma test remove *PragmaHandler* is called. The implementation of this function can be seen in line 19-22 in Listing 4.12. This function takes a *MatchObject* and an Insieme IR *StatementWrapper* (a list of IR statements) that contains the IR if statement. The function returns an empty list of statements. This means that the statements that are affected by the remove pragma are removed.

After the conversion of the *clang::ReturnStmt* into the corresponding IR statement, the function that is attached to the #pragma test change *PragmaHandler* is called. The implementation of this function can be seen in line 27-33 in Listing 4.12. This function takes a *MatchObject* that has to contain an IR expression for the variable that is used in the statement. The expression

is used to form a new IR return statement. This statement is packed into a *StmtWrapper* and returned. The *return 0;* is replaced by *return -1;*. After all pragmas are handled and the whole AST is converted into IR the target source code can be generated by the Insieme backend. The output source code for the example input code from Listing 4.10 can be seen in Listing 4.11. The if statement disappeared and the return statement returns -1 instead of 0.

```
1  /**
2   * ------------- Auto-generated Code -------------
3   * This code was generated by the Insieme Compiler
4   * -----------------------------------------------
5   */
6  #include <stdint.h>
7
8  /* ------- Function Definitions --------- */
9  int32_t main() {
10     return -1;
11 }
```

Listing 4.11: Pragma handling output code

4.6.4 Auto parallelization pragma example

This section provides an example of how user defined pragmas can be used to modify the IR in a useful way (e.g., provide automatic parallelization). The sample plugin from Listing 4.15 provides handling for two user defined pragmas:

- #pragma autopar barrier: A simple barrier that can be used to wait for all threads.
- #pragma autopar try_pfor: This pragma tries to execute a for loop in parallel. This means if the for loop is in a non-complex form (no free variables are used or modified inside the loop) the IR for statement will be converted into a parallel for statement. If the for loop uses or modifies free variables it won't be exchanged.

The input code that is shown in Listing 4.13 contains a for loop (line 17-19) that executes a function called *someExpensiveWork*, but without using free variables. Additionally, there is a second for loop (line 23-25) that uses free variables, and a barrier pragma (line 27). The first for loop will be identified as a non-complex for loop and will be converted into a parallel IR for loop by the frontend plugin. The second for loop will be identified as a complex for

```
#include "insieme/frontend/extensions/frontend_plugin.h"

using namespace insieme::frontend;
using namespace insieme::core;


insieme::core::NodeManager mgr;

class SamplePlugin : public extensions::FrontendPlugin {

public:
SamplePlugin() {

node&& remove_t = tok::eod;
node&& ret_t = tok::expr["return"] >> tok::eod;

//pragma: #pragma test remove
auto rem_pragma = PragmaHandler("test", "remove", remove_t,
    [](MatchObject object, stmtutils::StmtWrapper node) {
        //return empty statement
        return stmtutils::StmtWrapper();
    }
);

//pragma: #pragma test change return(-1)
auto change_pragma = PragmaHandler("test", "change", ret_t,
    [](MatchObject object, stmtutils::StmtWrapper node) {
        //get the new return expression
        ExpressionPtr retex = object.getExprs("return")[0];
        //build a new return statement
        ReturnStmtPtr stmt = ReturnStmt::get(mgr, retex);
        return stmtutils::StmtWrapper(stmt);
    }
);

//store the handlers in the PragmaHandler container
pragmaHandlers.push_back(
    std::make_shared<PragmaHandler>(rem_pragma)
);
pragmaHandlers.push_back(
    std::make_shared<PragmaHandler>(change_pragma)
);
}

};
```

Listing 4.12: sample pragma frontend plugin code

loop and won't be modified. The barrier pragma will be recognized and an IR barrier statement will be inserted before the call that prints the information that the execution has finished.

Analogous to the description of the user-provided pragma handling example (see Section 4.6.1) the locations of the pragmas are collected by the Clang preprocessor. The Clang preprocessor will recognize the three pragmas from the input code and will generate a map that is shown in Table 4.2. Again, this table contains the AST nodes and the list of pragmas that is connected to the Clang AST node. After the Clang AST to IR conversion of an affected node

AST node	**pragma**
clang::ForStmt@<MemoryAddress>	#pragma autopar try_pfor
clang::ForStmt@<MemoryAddress>	#pragma autopar try_pfor
clang::CXXOperatorCallExpr@<MemoryAddress>	#pragma autopar barrier

Table 4.2: Pragma map contents

(for statement or operator call expression) occurred, the lambda functions of the *PragmaHandlers* that are matching the mapped pragmas will be executed. The lambda functions are written in the plugin (see Listing 4.15). The function that will be called when converting an Clang AST node that is mapped to a #pragma autopar barrier element is written in line 14-23. The lambda function for #pragma autopar try_pfor is shown in line 26-42.

The generated IR code is shown in Listing 4.14. This code can be used to generate C/C++ code, that uses the Insieme Runtime system for a parallel execution. The generated IR shows that the first for loop was moved into a function that takes three arguments that define the range of the for statement and the step size. The original location of the for statement now contains a pfor IR element that will distribute the for loop iterations to the hardware.

```
#include <iostream>
#include <functional>
#include <string>

#define ITER 100000
#define N 1000

void someExpensiveWork() {
    std::string str = "Test string";
    std::hash<std::string> hash_fn;
    for(int i=0; i<ITER; i++)
        hash_fn(str);
}

int main() {
    #pragma autopar try_pfor
    for(int i=0; i<N; i++) {
        someExpensiveWork();
    }

    int k=0;
    #pragma autopar try_pfor
    for(int i=0; i<100; i++) {
        k += i;
    }

    #pragma autopar barrier
    std::cout << "Execution finished...\n";
    return 0;
}
```

Listing 4.13: Autopar pragma input code sample

4.7 Real-world plugin example

In this section a real-world plugin is presented. The plugin implements the features that are used to handle the conversion of in-line assembler statements into IR. This statements can be used as an extension of GCC and are used in the following way (like described in [21]):

- asm [volatile] (AsmTemplate : [OutpOpers] [: [InpOpers] [: [Clobbers]]])
- asm [volatile] goto (AsmTemplate : : [InpOpers] : [Clobbers] : Labels)

```
let fun000 = fun(int<4> v1, int<4> v2, int<4> v3) -> unit {
    for(decl int<4> v0 = v1 .. v2 : v3) {
        someExpensiveWork();
    };
};


let fun001 = fun() -> int<4> {
    pfor(getThreadGroup(0), 0, 100, 1,
      bind(v157, v158, v159){ fun000(v157, v158, v159) }
    );

    decl ref<int<4>> v165 =  var(0);
    for(decl int<4> v167 = 0 .. 100 : 1) {
        v165 :=  *v165+v167;
    };

    barrier();

    std::operator<<(RefIRToCpp(std::cout),
      ref.vector.to.src.array("Execution finished...\n"));
    return 0;
};


// Inspire Program
//  Entry Point:
fun001
```

Listing 4.14: Autopar pragma IR output code

```
#include "insieme/frontend/extensions/frontend_plugin.h"
#include "insieme/core/analysis/ir_utils.h"

using namespace insieme::frontend;
using namespace insieme::core;
NodeManager mgr;

class SamplePlugin : public extensions::FrontendPlugin {
public:
SamplePlugin() {
node&& eod_t = tok::eod;

//pragma: #pragma autopar barrier
auto barrier = PragmaHandler("autopar", "barrier", eod_t,
    [](MatchObject object, stmtutils::StmtWrapper node) {
        //get IRBuilder
        IRBuilder builder (mgr);
        //add an IR barrier before the statement-list
        node.insert (node.begin(), builder.barrier());
        //return the modified node
        return node;
    }
);

//pragma: #pragma autopar try_pfor
auto trypfor = PragmaHandler("autopar", "try_pfor", eod_t,
    [](MatchObject object, stmtutils::StmtWrapper node) {
        //assert it is a for stmt
        assert(node[0].isa<ForStmtPtr>());
        //check if we have no free variables
        bool free_v = analysis::hasFreeVariables(node[0]);
        //if there are free variables, no
        //simple auto parallelization is possible
        if(free_v)
            return node;
        //get IRBuilder
        IRBuilder builder (mgr);
        //build a parallel for statement
        node[0] = builder.pfor(node[0]);
        return node;
    }
);

//store the handlers in the PragmaHandler container
pragmaHandlers.push_back(
    std::make_shared<PragmaHandler>(barrier) );
pragmaHandlers.push_back(
    std::make_shared<PragmaHandler>(trypfor) );
}
};
```

Listing 4.15: Autopar pragma frontend plugin code

```
1 int src = 1;
2 int dst;
3
4 asm ("mov %1, %0\n\t"
5      "add $1, %0": "=r" (dst): "r" (src)
6     );
7
8 printf("%d\n", dst);
```

Listing 4.16: GCC assembler statement example code

- AsmTemplate: The string that contains the assembler code
- OutpOpers: A comma-separated list of the C variables modified by the assembler instructions
- InpOpers: A comma-separated list of C expressions read by the instruction
- Clobbers: A comma-separated list of registers or other values changed by the assembler instruction
- Labels: When using the goto form of asm, this section contains the list of all C labels to which the assembler instruction may jumps

Listing 4.16 shows two integer declarations and an assembler statement followed by a printf call. This example code from [21] copies *src* to *dst* and adds 1 to *dst*. The Insieme compiler supports the GCC assembler statements. The conversion into IR is implemented with the help of a frontend plugin (see Listing 4.17). The assembler frontend plugin is called before a Clang statement is converted, and if the Clang statement node is of type clang::AsmStmt the plugin is extracting the needed information (input operators, output operators, clobber, etc.) and is wrapping this information into an IR assembler statement. As a last step the converted element is returned.

```
#include "insieme/frontend/extensions/frontend_plugin.h"

using namespace insieme::frontend;
using namespace insieme::core;

class ASMPlugin : public extensions::FrontendPlugin {
public:

stmtutils::StmtWrapper Visit(const clang::Stmt* stmt,
    conversion::Converter& conv) {
 if(const clang::AsmStmt* st =
        llvm::dyn_cast<clang::AsmStmt>()) {
  //get IR Builder
  IRBuilder builder = conv.getIRBuilder();
  //create a string out of the assembler statement
  std::string asmStr = st->generateAsmString(conv.
     getCompiler().getASTContext());

  //create an IR assembler statement
  lang::AsmStmtWrapper wrap (asmStr, st->isVolatile());

  //get output elements and add the
  //converted expressions to the IR statement
  for (unsigned i=0 ; i<st->getNumOutputs(); ++i) {
    wrap.addOutput (
     st->getOutputConstraint(i).str(),
     builder.deref(conv.convertExpr(st->begin_outputs()[i]))
    );
  }

  //get input elements and add the
  //converted expressions to the IR statement
  for (unsigned i=0 ; i<st->getNumInputs(); ++i) {
    wrap.addInput (
     asmStmt->getInputConstraint(i).str(),
     conv.convertExpr(st->begin_inputs()[i])
    );
  }

  //add the clobber information
  for (unsigned i=0 ; i< st->getNumClobbers (); ++i) {
    wrap.addClobber(st->getClobber(i).str());
  }

  //return the converted assembler statement
  return lang::toIR(conv.getNodeManager(), wrap)
 }
 return nullptr;
}
};
```

Listing 4.17: GCC assembler statement plugin code

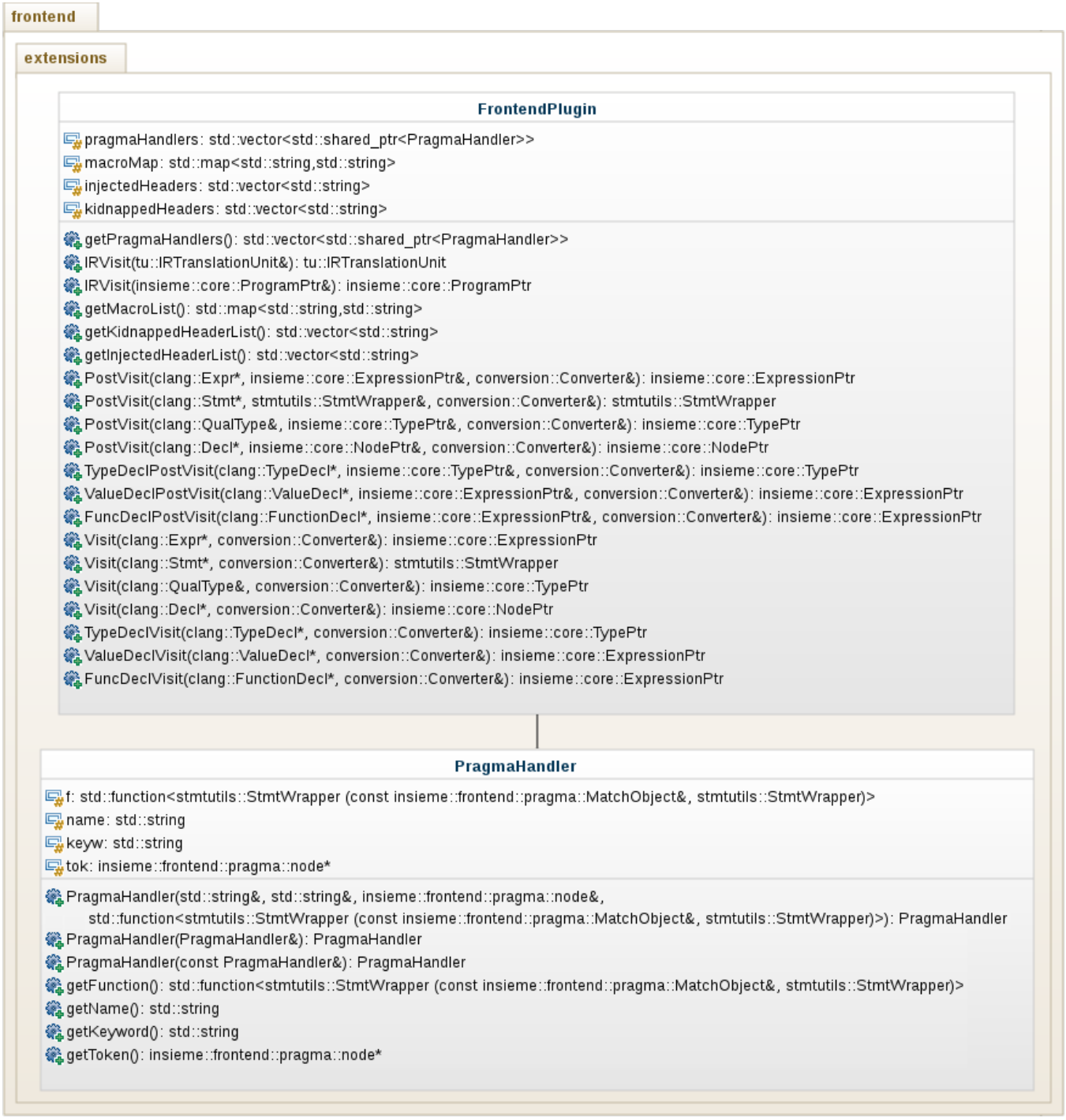

Figure 4.12: Full class diagram of the Insieme FrontendPlugin class

Chapter 5

Conclusion and Future work

The development of a compiler can be seen as a challenging work that requires excellent programming skills, knowledge of programming languages and the theory behind it. Especially for high level languages, like C++, the development phase will never be completely finished, due to the reason that a new language standard is released every three to five years.

The first part of this thesis discussed the frontend implementation of the Insieme compiler, and the new interface to the compiler (named insiemecc) that helps the end-user (that wants to compile source code) to use the Insieme compiler in an efficient and easy way. It is possible now to use the Insieme compiler infrastructure to compile code bases of arbitrary sizes. The insiemecc driver application can also be used to replace GCC, LLVM, or any other C/C++ compiler (e.g., compiler used in a Makefile).

The second part explained the features of the newly developed frontend plugin system. The main objective of the frontend plugin system is to create an interface that can be used by the compiler developers that are working on the Insieme project. The plugin system helps compiler developers to implement new features in a fast, secure, efficient, and simple way. It is not needed to change the frontend core source files anymore, and therefore the error-proneness of the frontend is decreased.

The Insieme compiler project is an ongoing effort and therefore there are many topics for future work. This work aimed to create a more sophisticated way to use the Insieme compiler for both the compiler developers and the users that want to compile source code with the Insieme compiler. The results of this work open the following future work topics:

- Dynamic linking of plugins: At the moment it is necessary to register the frontend plugins by hardcoding the registration call in the Insieme frontend. This is not very user-friendly and should be done with dynamic

linking or by providing a special flag when calling insiemecc. The LLVM pass infrastructure provides a convenient interface for user provided functionalities [8].

- Interfering plugins: It may happen that plugins are interfering. This can for example be the case when two different plugins are acting in the same frontend phase (e.g., two plugins are modifing the IRProgram). To avoid mistranlations and bugs that are introduced by plugins it would be convenient to have some kind of priority list or plugin dependency graph to define the plugin execution order.

- Insiemecc support for more flags: Currently insiemecc doesn't provide full support for all flags that can be used when using GCC or LLVM. To avoid compilation errors because of unsupported flags insiemecc is ignoring unknown flags at the moment. Nevertheless, support for the missing flags should be implemented or at least passed to the backend compiler in order to avoid malformed results.

- Missing C++ features and new language standards: The Insieme compiler doesn't support all C++ features. Especially the C++11 support is only partially implemented and there is no support for C++14 features yet.

List of Figures

List of Tables

List of Listings

Bibliography

[1] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Elsevier, Amsterdam, 5th edition, 2012.

[2] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[3] Radu Prodan. Compiler construction 2013 [slides].

[4] Wikipedia entry for Compiler.
`http://de.wikipedia.org/wiki/Compiler#Arbeitsweise`.
Accessed: 2014-07-25.

[5] clang website.
`http://clang.llvm.org`.
Accessed: 2014-07-25.

[6] clang website - Features and Goals.
`http://clang.llvm.org/features.html`.
Accessed: 2014-07-25.

[7] ROSE compiler infrastructure.
`http://rosecompiler.org/`.
Accessed 2014-10-25.

[8] Writing an LLVM Pass.
`http://llvm.org/docs/WritingAnLLVMPass.html`.
Accessed: 2014-10-25.

[9] Plugin API - GCC Internals.
`https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html`.
Accessed: 2014-10-25.

[10] Cetus documentation - Writing a Pass.
`http://cetus.ecn.purdue.edu/Documentation/manual/ch09.html`.
Accessed: 2014-10-25.

[11] Insieme compiler project.
http://www.insieme-compiler.org.
Accessed: 2014-09-01.

[12] 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.

[13] Dr. Peter Thoman. *Insieme-RS - A Compiler-supported Parallel Runtime System*. PhD thesis, University of Innsbruck, 2013.

[14] Insieme compiler architecture description.
http://www.insieme-compiler.org/architecture.html.
Accessed: 2014-09-01.

[15] clang API documentation.
http://clang.llvm.org/doxygen/classclang_1_1CompilerInstance.html.
Accessed: 2014-07-25.

[16] Wikipedia entry for C preprocessor.
http://en.wikipedia.org/wiki/C_preprocessor.
Accessed: 2014-08-29.

[17] Bernhard Höckner. Master's thesis, University of Innsbruck, 2014.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Pearson Education, Amsterdam, 1. edition, 1994.

[19] Wikipedia entry for The Visitor Pattern.
http://en.wikipedia.org/wiki/Visitor_pattern.
Accessed: 2014-09-11.

[20] Dr. Simone Pellegrini. *An experimental framework for Pragma Handling in Clang*. Talk given at European LLVM Conference, July 2013.

[21] Assembler Instructions with C Expression Operands.
https://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/Extended-Asm.html.
Accessed 2014-10-28.