Insieme’s Haskell-based Analysis Toolkit

Putting a HAT on Insieme

Alexander Hirsch

30 April 2017

Table of Contents

Abstract 5

1 Introduction 6
- 1.1 Attribution 6
- 1.2 Motivation 6
- 1.3 Objectives 6
- 1.4 Picking Haskell 6
- 1.5 Overview 7

2 Static Code Analysis 8
- 2.1 Purpose 8
- 2.2 Control-Flow Graph 8
- 2.3 Data-Flow Analysis 9
  - 2.3.1 Lattice 9
  - 2.3.2 Reaching Definitions Example 10
    - 2.3.2.1 Lattice 10
    - 2.3.2.2 Data-Flow Analysis 10
- 2.4 Constraint-Based Analysis 11
  - 2.4.1 Reaching Definitions Example (cont) 11
- 2.5 Insieme CBA framework 11
  - 2.5.1 Property Space 12
  - 2.5.2 Analysis Variables 12
  - 2.5.3 Assignment 13
  - 2.5.4 Constraint Guards 13
  - 2.5.5 Constraint Generator 13
  - 2.5.6 Analysis 13
  - 2.5.7 Constraint Solver 13

3 Architecture 14
- 3.1 Insieme 14
- 3.2 INSPIRE 14
  - 3.2.1 Structure 14
    - 3.2.1.1 Notation 15
    - 3.2.1.2 Nodes 15
    - 3.2.1.3 Value Nodes 15
    - 3.2.1.4 Type Nodes 15
    - 3.2.1.5 Statement Nodes 16
    - 3.2.1.6 Expression Nodes 16
    - 3.2.1.7 Support Nodes 16
    - 3.2.1.8 Record Nodes 16
- 3.3 Extension Mechanism 17
  - Operators and Analyses 17
  - 3.3.1 The Basic Language Extension 17
    - 3.3.1.1 Identifiers 18
    - 3.3.1.2 Arithmetic Types and Operators 18
    - 3.3.1.3 Boolean Types and Operations 18
  - 3.3.2 Reference Extension 18
  - 3.3.3 Parallel Extension 19

3.4 Semantics 19
3.4.1 Empty Program 19
3.4.2 Variable Declaration 20
3.4.3 Basic Arithmetic 20
3.4.4 Function Call 21
3.4.5 For Loop 22
3.5 Haskell-based Analysis Toolkit 23
3.5.1 Adapter 24
3.5.2 Utilities 24
3.5.3 INSPIRE 24
3.5.4 Framework 24
3.5.5 Base Analyses 25
3.5.6 User Analyses 25

4 Framework Implementation 26
4.1 Build Process 26
4.1.1 Compiling HAT into a Library 26
4.1.2 Linking the Shared Libraries 26
4.2 Adapter 27
4.2.1 Foreign Function Interface 27
4.2.2 Haskell RunTime System 28
4.2.3 Context 28
4.3 IR Components 28
4.3.1 Node Types 28
4.3.2 Node Path 29
4.3.3 Binary Parser 29
4.3.4 Visitors 31
4.3.5 Node References and Queries 31
4.3.6 Context 31
4.4 Utilities 32
4.4.1 Bound Sets 32
4.4.2 Arithmetic Formulas 33
4.5 Fixpoint Solver 34
4.5.1 Lattice 34
4.5.2 Variables 34
4.5.3 Assignments 35
4.5.4 Constraints 35
4.5.4.1 Dependencies between Variables 36
4.5.5 Solver 36
4.6 Modelling Arrays, Structs, and Unions 36
4.7 Generic Data-Flow Analysis 38
4.7.1 Operator Handler 38
4.8 Program Point 39
4.8.1 Program Point Analysis 39
4.8.1.1 Data-Flow vs Program Point 39
4.9 Memory State 40
4.9.1 Reaching Definitions Analysis 40
4.9.2 Memory State Analysis 40
4.10 Specific Analyses 41
4.10.1 Identifier Analysis 41
4.10.1.1 Connection Between Generic DFA and Specialised Analysis 42

4.10.2 Arithmetic Analysis 42
4.10.3 Boolean Analysis 45
4.10.4 Data Path Analysis 46
4.10.5 Reference Analysis 47
4.10.6 Callable Analysis 48
4.11 Helper Analyses 49
4.11.1 Reachability Analysis 49
4.11.1.1 Reachable In 49
4.11.1.2 Reachable Out 50
4.11.2 Exit Point Analysis 51
4.11.3 Predecessor Analysis 51
4.11.4 Call Site Analysis 52

5 Constructing a new Analysis 53
5.1 Defining the Goals 53
5.2 Investigating INSPIRE 53
5.3 Designing the Analysis 56
5.4 Defining the Property Spaces 57
5.5 Defining the Interface 57
5.6 Setup Boilerplate 57
5.7 Write Test Cases 58
5.8 Implementation 58
5.8.1 Element Count Analysis 58
5.8.2 Getting the Array Index 59
5.8.3 Putting it Together 60
5.9 Summary 60

6 Evaluation 61
6.1 Qualitative Evaluation 61
6.2 Quantitative Evaluation 62

7 Conclusion 64
7.1 Future work 64

8 Appendix A: INSPYER 65
8.1 JSON Export 65
8.2 User Interface 66
8.3 Meta Data 67

9 Appendix B: Evaluation Data 69

References 70

Abstract

This document provides an insight into the Haskell-based Analysis Toolkit (HAT), allowing developers and researchers to rapidly prototype and develop static program analyses. It is therefore composed of an introduction into the topic of static program analysis, an architectural overview of the framework, the specification of a variety of essential analyses and components, a tutorial for designing new analyses showcasing the framework, followed by an evaluation of the frameworks capabilities. The conclusion summarises the contributions of this thesis and provides an outlook on future work.

For the tutorial, and to showcase the framework, an array out-of-bounds analysis is constructed. This analysis is also utilised for evaluating the framework, by investigating its ability of implicitly expanding a program analysis language feature support. Furthermore we evaluate the frameworks performance in terms of execution time and memory requirements by analysing 63 622 different properties within a total of 320 example codes.

The complete code of this project is available on GitHub embedded into the high-level analysis module of the Insieme compiler.

1 Introduction

This document describes a framework for easing the implementation of data-flow analyses within the Insieme project. Note that throughout this document the terms framework, toolkit, and Haskell-based Analysis Toolkit (HAT) are used interchangeably.

The core contribution of this project is the implementation of a scalable, easy-to-use static program analysis framework, which can be used for rapid development and prototyping of data-flow analyses.

As this project is a joint effort between Herbert Jordan, Thomas Prokosch, and myself, a section dedicated to the attribution is provided next. It is followed by a motivation section, the objectives, and a justification for picking Haskell as platform for the framework. The introduction is concluded with a structural overview of the remaining document.

1.1 Attribution

The theoretical groundwork, as well as a first prototype has been created by Herbert Jordan, who also plays a major role in the current development of the framework. He is tasked with implementing the core components of the framework and overlooking the development process.

Thomas Prokosch helped with the initial technology assessment and prototyping components in Haskell before they are integrated into the framework.

The main tasks of the author of this thesis are:

- integration of the C++ / Haskell interface
- framework interface design
- Intermediate Representation (IR) data structure
- documentation and tutorial
- implementing utilities
- example analyses
- debugging tools

1.2 Motivation

Program analysis, in general, is a big topic, not only in compiler theory. Extracting certain properties automatically from an input program can be viable for multiple different use-cases (eg auto-tuning or code validation). When talking about compilers, many of their optimisations rely on some kind of Data-Flow Analysis (DFA) to identify potential optimisation options.

The biggest nugget of motivation can be mined from the idea of having a generic Constraint-Based Analysis (CBA) framework at hand, which already provides a basic set of common analyses and can be extended rapidly as needed. The generic DFA provided by the framework presented by this thesis handles (among others) declarations, parameter and return value passing, closures, support for composed data types, calls to unknown external functions, inter-procedural analyses, and memory locations.

By integrating it into a source-to-source compiler, and operating on a high-level intermediate representation, little boilerplate code is required to construct new analyses for various tasks. Prototypes for research as well as for development purposes can be efficiently implemented.

1.3 Objectives

The ultimate goal of this project is the realisation of a scalable implementation of the CBA framework developed by Jordan (2014) exhibiting superior usability. The main objective of this thesis is to help lay the foundation for this framework and its integration with Insieme. To achieve the ultimate goal, further development will build upon this work in the future.

The main objective therefore consists of developing the core components of the framework. These are composed of a *fixpoint solver*, a smaller set of generic analyses, and a bigger set of basic (more specific) analyses. The intention behind each of these components and their relationship with each other is explained in Framework Implementation.

The framework is a part of the Insieme project and therefore operators on the IR of the Insieme compiler – namely the INSieme Parallel Intermediate REpresentation (INSPIRE). This IR is implemented as C++ data structure in the core of the compiler. However, as the framework is not realised using C++ (see below), a suitable representation of INSPIRE needs to be established. This also includes a way to transfer an INSPIRE program from the Insieme compiler to this representation. Furthermore, the framework should provide a way of relaying back analysis results to C++.

Also, it should be possible to use the framework as standalone, without the need for the Insieme compiler.

Last but not least, creating documentation of the framework and a manual for further analysis development is needed. This objective is reflected by this document.

1.4 Picking Haskell

The first prototype of the CBA framework has been implemented using C++. During its development drawbacks of using C++ manifested in the form of long compile times yielding a very slow development cycle. Due to this reason, a different programming language (platform) has been chosen for the framework.

Due to the mathematical nature of program analysis, a functional programming language is preferred for the implementation. The mathematical constructs defining analyses can be realised much simpler via the use of Algebraic Data Structures (ADTs) offered by functional languages, compared to structs, classes, and records of imperative and object-oriented languages.

Haskell, popular amongst functional programming languages, features expressive syntax, a strong, static type system, and a growing community maintaining thousands of packages and libraries. It was therefore ultimately selected for this task. The average reader of this document may not be familiar with Haskell – or functional programming languages in general – and is therefore invited to pick it up and start their functional programming adventure. Lipovaca (2011) composed an excellent book going by the name of *Learn You a Haskell for Great Good!: A Beginner's Guide*, endorsed by the Haskell community, and available online. Core concepts are explained with additional background information and useful analogies. The playful art style and pop culture references positively influence the learning experience. We also recommend the books *Real World Haskell* (O'Sullivan, Goerzen, and Stewart 2008), *Haskell Design Patterns* (Lemmer 2015), and *Haskell High Performance Programming* (Thomasson 2016).

Other functional programming languages like OCaml, F#, Scala, and Erlang were considered too, yet have been ruled out during an initial technology assessment phase. Since F# builds on the .NET framework it was quickly removed from the list of considerations. The overall footprint seems too large and no benefits of having the .NET framework on-board materialised. Both Scala and Erlang seem interesting as well as promising, although on further investigation Haskell promises a more advanced type system and people have reported that its easier applicable and better designed compared to Scala (Nordenberg 2012). In contrast to Erlang, Haskell is statically typed – something we very much prefer as it makes it easier to catch bugs early in the development process. OCaml, in fact quite similar to Haskell, has been reconsidered multiple times during the prototyping phase. In the end, personal preference in the syntax and design of the standard library resulted in the selection of Haskell instead of OCaml.

For the remainder of this thesis, it is assumed that the reader has a basic understanding of the Haskell programming language. Nevertheless, details and more complex concepts are introduced when needed, along with additional references.

1.5 Overview

Static Code Analysis provides a brief introduction to static code analysis, focusing on *flow-sensitive* analysis. The concept is communicated via an example utilising a more conventional DFA framework, followed by the constraint-based approach. This leads to a generic CBA framework, which is originally based on the one presented by Nielson, Nielson, and Hankin (1999). Additionally, customisations and adaptations towards integrating CBA capabilities into Insieme have been made by Jordan (2014) to the framework, which not only seem promising in theory but also bear in mind real-world use-cases and practical applications – something not encountered that often in the field of program analysis.

Next, covered by Architecture the overall architecture of Insieme's Haskell-based Analysis Toolkit (HAT) is depicted, delivering the big picture of the task at hand. This includes presenting the Insieme compiler followed by a detailed investigation of INSPIRE Jordan et al. (2013).

The implementation part, described in Framework Implementation, reflects the documentation of the current state of the framework. The source code can be found on GitHub embedded into the analysis module of the Insieme compiler. Presented code snippets have been simplified for clarity in some cases.

In Constructing a new Analysis a new analysis is constructed showcasing the new framework. Furthermore, the newly implemented analysis is used to evaluate the framework in Evaluation. Finally, Conclusion summarises the contributions of this work an provides an outlook on future development.

2 Static Code Analysis

This chapter provides a concise introduction to the topic of static program analysis with a focus on Data-Flow Analysis (DFA) and its extension, the *constraint-based approach*. It serves as a stepping stone to communicate the theoretical background for the following chapters.

It starts off with a few words about the general purpose of static code analysis and continues with the introduction of the Control-Flow Graph (CFG). After that the general concept of DFA is communicated together with a *reaching definitions* example. Finally, the constraint-based approach is tackled and applied to the same example for comparison. The chapter concludes with an overview of customisations made by Jordan (2014) to the Constraint-Based Analysis (CBA) framework.

2.1 Purpose

Static code analysis is about deducing *properties* of a program without actually running the program. This happens typically in a compiler during the optimisation stage. Yet there are also other tools (referred to as static code analysers) which can run various checks on the code and extract properties from it. Checking for nullpointer dereferences, dead code, and estimating runtime complexity are just a few examples for checks, that can be conducted by an analysis. In case of a compiler, an optimisation commonly consists of an analysis identifying structures in the program suitable for optimisation, followed by a transformation of that structure into one that yields better performance[1]. Static code analysers are often used to identify bugs (eg race conditions, iterator invalidation, etc) not caught by the compiler. But they can also be useful to extract program properties for later use (eg auto-tuning).

As already mentioned, this introduction section focuses on DFA, which is a *flow-sensitive* technique. Meaning, we take the control-flow of the input program into account at the time the analysis is performed. This leads to a more accurate result, but also requires a greater computational effort. *Flow-insensitive* analyses, on the other hand, are much faster. However, they yield less accurate results in the form of over-approximations. Since knowledge about flow-insensitive analysis is not required for this task, these kinds of techniques are not elaborated further.

There are, of course, different techniques how one could construct a DFA. And while some of them provide clear benefits over others, they sometimes also suffer drawbacks. But what (nearly) all of them have in common is the utilisation of CFGs. The following section introduces those.

2.2 Control-Flow Graph

The control flow of a program can be abstracted as a graph, in which each node represents a *basic block*. A basic block is a sequence of instructions without any jumps or jump targets (ie labels) in-between. It commonly begins with a jump target and ends with a jump instruction. Basic blocks are connected via directed edges, which represent potential execution paths. An edge that points to an already visited block during depth-first traversal is referred to as a *backward edge* and frequently introduced by a loop. Adding two additional empty blocks, the *entry block* and *exit block*, can help keeping algorithms operating on the graph simple. All control-flows start at the entry block and end at the exit block (Wikipedia 2017a).

What exactly constitutes a node of the CFG is not set in stone and varies depending on the application. While the textbook definition concerns itself with basic blocks, one could easily abstract away whole functions into a graph's nodes. Based on the application, attaching additional information to the edges can be helpful too.

The code snippet below displays a short C program with three functions. Figure 1 illustrates the corresponding CFG. Each function features a dedicated entry node annotated with the function's name and an exit (`return`) block. The example contains branching as well as loops.

```c
/* File: cfg.c */

#include <stdio.h>
#include <stdlib.h>

void print_something(int times) {
    for (int i = 0; i < times; ++i) {
        puts("something");
    }
}

void print_usage() {
    puts("some usage");
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        print_usage();
        return EXIT_FAILURE;
    }
    print_something(atoi(argv[1]));
    return EXIT_SUCCESS;
}
```

[1] Depending on the current objective, performance can refer to runtime, memory usage, power consumption, etc.

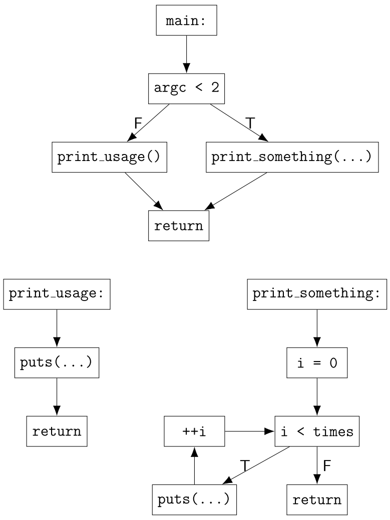


Figure 1: A CFG corresponding to `cfg.c`.

During a program analysis run certain properties are tracked across the nodes. These properties (eg reaching definitions) are recorded when a node is entered and left. These records are referenced by the labels of the related graph node together with either in or out.

For instance, $\mathrm{REACH}^{\mathrm{in}}[S]$ and $\mathrm{REACH}^{\mathrm{out}}[S]$ are the reaching definitions recorded at the entry point and exit point respectively of the graph node$S$. This notation is often shortened to $\mathrm{in}(S)$ / $\mathrm{out}(S)$ if the specific analysis can be derived from the current context(Wikipedia 2016b).

2.3 Data-Flow Analysis

In this section the basis of Data-Flow Analysis is described. The description itself is keep very informal and is presented through an example.

A Data-Flow Analysis (DFA) is a static program analysis used to extract certain properties (eg *reaching definitions* or *live variables*) from a program. Traditionally, DFA assumes a given control-flow of the program – and while one could work on the Abstract Syntax Tree (AST) – using a CFG is often much simpler and yields similar outcome (with respect to accuracy regarding DFA) (Chong 2011). This technique can be cast into a framework as presented in the remainder of this document.

Furthermore, there are two ways an analysis can operate. It can either be a *must analysis* or a *may analysis*. A must analysis requires that a property must hold in all branches to be still considered after a meeting point (logical and). Alternatively, in a *may analysis* a property is considered after a meeting point if it holds in at least one branch (logical or). An example for a *must analysis* would be *available expressions* (Chong 2011), while *reaching definitions* (see Reaching Definitions Example) is a *may analysis*.

Most DFA frameworks only yield sound solutions if the domain of the property space forms a *lattice*. Because of this, we now take a short detour and glance at the mathematical concept of a lattice.

2.3.1 Lattice

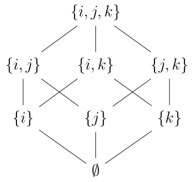


Figure 2: A Hasse diagram of an example lattice.

A complete lattice is an algebraic structure which consists of a partially ordered set in which every two elements share a unique upper bound (*join*) and a unique lower bound (*meet*). Furthermore there exists one element $\perp$ that is smaller, and one element $\top$ that is larger than every other element in the set. The purpose of the lattice, with respect to program analysis, is to model a domain of program properties. The *height* of a lattice is given by the longest path through partial order from greatest to least. Elements of the lattice represent flow values (in and out sets) while $\perp$ and $\top$ represent the *best-case* and *worst-case* information respectively. The *meet* operator $\sqcup$ describes how two flow values are merged (Wikipedia 2016a; Chambers 2006; Strout 2006).

Figure 2 shows an example application for a lattice where three variables i, j and k are given. A flow value can exhibit any combination of them. Hence the $\top$ and $\perp$ value can be mapped to the set of all variables and $\emptyset$ respectively. The height of the lattice, 4, which can be immediately deduced from the illustration and the ordering of the lattice ($\sqsubseteq$) is mapped to $\subseteq$. This representation is known as a *Hasse Diagram* (Wikipedia 2017b).

There exists also the concept of a *semilattice*, which resembles a normal lattice, but only features *either* a least upper bound *or* greatest lower bound, known as *join-semilattice* or *meet-semilattice*, respectively.

The interested reader is encouraged to take a look at the more comprehensive introduction to lattices provided byJovanovic (2005).

2.3.2 Reaching Definitions Example

A definition of a variable at program point p^1 *reaches* program point p^2 if there is a control-flow path from p^1 to p^2 along which the variable defined by p^1 is not redefined. Positive and negative examples are provided (Wikipedia 2016b).

```
p1: x = 42                                            p4: z = 24
p2: y = 12                                            p5: z = 16
p3                                                    p6
```

The definition of x from p^1 reaches the program point p^3. The definition of z from p^4 does not reach p^6. It is by the definition at p^5.

2.3.2.1 Lattice

Before concerning ourselves with the DFA itself, we have to define the lattice of the property we would like to obtain. Let V be the set of all variables, $S = \{0, 1, \ldots\}$ the set of all statements in the CFG and $L = S \cup \{?\}$. The lattice is then given by

$$(2^{V \times L}, \subseteq)$$

'$?$' is used to represent an input to the program. From this, $\bot = \emptyset$ and $\top = V \times L$ can be derived. Since the elements of V and L are ordered, we can also derive an order for $V \times L$. The join and meet operators can be mapped to $\cup$ and $\cap$ respectively.

2.3.2.2 Data-Flow Analysis

The data-flow equations for calculating reaching definitions are: for all basic blocks b of a given CFG,

$$\begin{aligned} \mathrm{in}(b) &= \bigcup_{p \in \mathrm{pred}(b)} \mathrm{out}(p) \\ \mathrm{out}(b) &= \mathrm{gen}(b) \cup (\mathrm{in}(b) \setminus \mathrm{kill}(b)) \end{aligned}$$

where $\mathrm{gen}(b)$ is the set of definitions contained in block b, $\mathrm{kill}(b)$ the set of definitions *killed* in block b and $\mathrm{pred}(b)$ is the set of predecessors of b. *Killing* a definition means that it is no longer available in subsequent blocks, as illustrated in the initial example of this subsection.

Let us take the following program computing $x!$ together with the corresponding CFG in Figure 3. Note that we only focus on the calculating part here and assign each statement its own block for simplicity. This example is based on the introduction provided by Nielson, Nielson, and Hankin (1999).

```c
/* File: factorial.c */

#include <stdio.h>
#include <stdlib.h>

int factorial(int x) {
    int y = x;
    int z = 1;
    while (y > 1) {
        z = z * y;
        y = y - 1;
    }
    y = 0;
    return z;
}

int main(int argc, char* argv[]) {
    int x = atoi(argv[1]);
    printf("%d! = %d\n", x, factorial(x));
    return EXIT_SUCCESS;
}
```

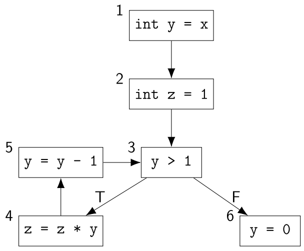


Figure 3: A CFG corresponding to `factorial.c`.

Running a reaching definition analysis on this code results in the following twelve equations, two for each block.

$$\begin{aligned} &\mathrm{in}(1) = \{(x, ?)\} && \mathrm{out}(1) = \{(y, 1)\} \cup (\mathrm{in}(1) \setminus \{(y, l) \mid l \in L\}) \\ &\mathrm{in}(2) = \mathrm{out}(1) && \mathrm{out}(2) = \{(z, 2)\} \cup (\mathrm{in}(2) \setminus \{(z, l) \mid l \in L\}) \\ &\mathrm{in}(3) = \mathrm{out}(2) \cup \mathrm{out}(5) && \mathrm{out}(3) = \mathrm{in}(3) \\ &\mathrm{in}(4) = \mathrm{out}(3) && \mathrm{out}(4) = \{(z, 4)\} \cup (\mathrm{in}(4) \setminus \{(z, l) \mid l \in L\}) \\ &\mathrm{in}(5) = \mathrm{out}(4) && \mathrm{out}(5) = \{(y, 5)\} \cup (\mathrm{in}(5) \setminus \{(y, l) \mid l \in L\}) \\ &\mathrm{in}(6) = \mathrm{out}(3) && \mathrm{out}(6) = \{(y, 6)\} \cup (\mathrm{in}(6) \setminus \{(y, l) \mid l \in L\}) \end{aligned}$$

Each pair consists of a variable (x, y or z) and a label, where the label is either the number of the block or $?$’ if the variable is not initialised within this context. Uninitialised variables can be considered input to the program. When focusing on block 3 one can see that the information flowing into the block is the same as the information flowing out of the block since no variables are modified. At block 3 two different control-flows meet yielding a merge of facts.

One can observe a different result for the first equation when comparing it to the original source of this example. This change results from the way this example is presented. In our case the two variables y and z do not exist prior to executing the algorithm, hence the analysis does not need to consider them as input (Nielson, Nielson, and Hankin 1999, 7).

As stated by Nielson, Nielson, and Hankin (1999), only a finite number of solutions exist for the derived system of equations. Furthermore each of these solutions is a *fixed point* and all of them are partially ordered. Most interestingly, the *least fixed point solution*, obtained this way, contains the fewest pairs of reaching definitions. Therefore this is the most accurate result, and the one we are interested in. It looks as follows:

$$\begin{array}{ll}
\mathrm{in}(1) = \{(x,?)\} & \mathrm{out}(1) = \{(x,?),(y,1)\} \\
\mathrm{in}(2) = \{(x,?),(y,1)\} & \mathrm{out}(2) = \{(x,?),(y,1),(z,2)\} \\
\mathrm{in}(3) = \{(x,?),(y,1),(y,5),(z,2),(z,4)\} & \mathrm{out}(3) = \{(x,?),(y,1),(y,5),(z,2),(z,4)\} \\
\mathrm{in}(4) = \{(x,?),(y,1),(y,5),(z,2),(z,4)\} & \mathrm{out}(4) = \{(x,?),(y,1),(y,5),(z,4)\} \\
\mathrm{in}(5) = \{(x,?),(y,1),(y,5),(z,4)\} & \mathrm{out}(5) = \{(x,?),(y,5),(z,4)\} \\
\mathrm{in}(6) = \{(x,?),(y,1),(y,5),(z,2),(z,4)\} & \mathrm{out}(6) = \{(x,?),(y,6),(z,2),(z,4)\}
\end{array}$$

2.4 Constraint-Based Analysis

Previously we dealt with the extraction of multiple equations from a CFG by traversing it, and then solving a system of equations to derive information (ie facts) about the program. The main idea behind the constraint-based approach is to directly extract these constraints and handle them explicitly, instead of implicitly handling equations given by the structure of the CFG. Due to the explicit handling of these constraints the two processes of generating and solving them can be separated, resulting in a *constraint generator* and *constraint solver* (Nielson, Nielson, and Hankin 1999; Jordan 2014).

We continue straight on with the example of the previous section to demonstrate the differences between the mentioned two approaches.

2.4.1 Reaching Definitions Example (cont)

Taking the constraint-based approach, the following list of constraints can be derived from the previous example – see Figure 3. Note that the format of constraint may be more complex (eg conditions), depending on the scenario. They may also be derived from other sources than the CFG (eg an AST).

$$\begin{array}{ll}
\mathrm{out}(1) \supseteq \mathrm{in}(1) \setminus \{(y,l) \mid l \in L\} & \mathrm{out}(1) \supseteq \{(y,1)\} \\
\mathrm{out}(2) \supseteq \mathrm{in}(2) \setminus \{(z,l) \mid l \in L\} & \mathrm{out}(2) \supseteq \{(z,2)\} \\
\mathrm{out}(3) \supseteq \mathrm{in}(3) & \\
\mathrm{out}(4) \supseteq \mathrm{in}(4) \setminus \{(z,l) \mid l \in L\} & \mathrm{out}(4) \supseteq \{(z,4)\} \\
\mathrm{out}(5) \supseteq \mathrm{in}(5) \setminus \{(y,l) \mid l \in L\} & \mathrm{out}(5) \supseteq \{(y,5)\} \\
\mathrm{out}(6) \supseteq \mathrm{in}(6) \setminus \{(y,l) \mid l \in L\} & \mathrm{out}(6) \supseteq \{(y,6)\}
\end{array}$$

Upon close investigation of this list and the original program, a pattern emerges. Every time a variable is assigned, two constraints are generated. One for excluding (*killing*) all pairs generated by prior assignments of this variable, and one for adding (*generating*) a new pair corresponding to this assignment. Otherwise, if no variable is assigned, everything is simply passed through.

Note the similarities of this discovered pattern and the original dataflow equations:

$\mathrm{out}(b) = \mathrm{gen}(b) \cup (\mathrm{in}(b) \setminus \mathrm{kill}(b))$	Dataflow equation
$\mathrm{out}(b) \supseteq \mathrm{gen}(b)$ $\mathrm{out}(b) \supseteq \mathrm{in}(b) \setminus \mathrm{kill}(b)$	Constraints

Nevertheless, these constraints only tell us how information flows *through* each block, we are still missing how information flows *between* blocks. Therefore the following list is derived.

$$\begin{array}{ll}
\mathrm{in}(2) \supseteq \mathrm{out}(1) & \\
\mathrm{in}(3) \supseteq \mathrm{out}(2) & \mathrm{in}(3) \supseteq \mathrm{out}(5) \\
\mathrm{in}(5) \supseteq \mathrm{out}(4) & \\
\mathrm{in}(6) \supseteq \mathrm{out}(3) &
\end{array}$$

Again a pattern is immediately visible: each edge in the CFG results in a constraint connecting the exit set of a block with the entry set of its successor.

The last constraint we need relates to the input of our program.

$$\mathrm{in}(1) \supseteq \{(x,?)\}$$

Finally, the same solution obtained in the data-flow analysis approach is also valid for this system of constraints. In fact, it is also the least solution (Nielson, Nielson, and Hankin 1999, 10).

2.5 Insieme CBA framework

The constrained-based approach, as presented by Nielson, Nielson, and Hankin (1999), resembles a generic framework suitable for various kinds of analysis. While we only focused on reaching definitions for now, other kinds (eg live variable analysis) can be realised in a similar fashion.

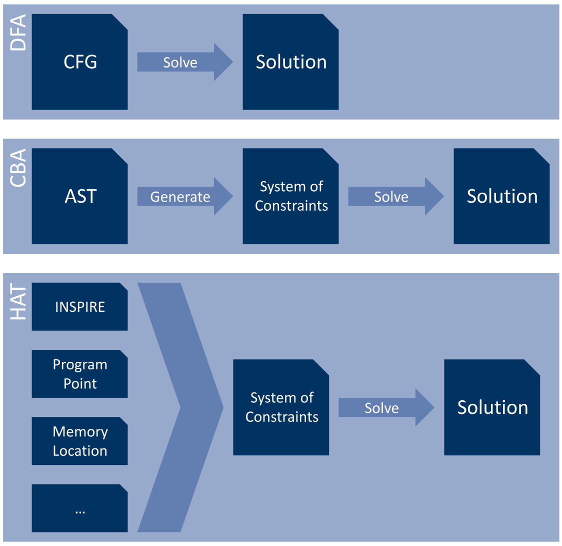


Figure 4: Illustrating the difference between DFA, CBA, and HAT.

In Figure 4 the core difference between DFA, CBA and our HAT are illustrated. In DFA a system of equations is implicitly derived from the CFG and immediately solved to obtain a valid solution for an analysis. In CBA the generation and solving of constraints is replaced by two stages: first generating a system of constraints and solving the system afterwards. Both stages are clearly separated and the second is initiated *after* the first one is completed. HAT on the other hand, relaxes the strict design of two separated stages. This is made possible by the introduction of a lazy constraint solver (Jordan 2014).

The lazy constraint solver allows the toolkit to generate constraints on the fly, as needed, and to inject them into the system of constraints during an analysis run. Furthermore, HAT is not limited to using an (annotated) CFG. As already hinted in the introduction to this chapter, not all analyses operate on a CFG. Instead, they could directly use an AST or their own custom data structure.

Using the INSieme Parallel Intermediate REpresentation (INSPIRE) together with additional sources of information, including but not limited to Program Points and Memory Locations, analyses can be more flexible and can output more accurate results than by just using a CFG. In general, the framework allows multiple different structures to be associated with *analysis variables*. Also, the extraction of a CFG from an AST can be skipped. In particular, this design enables the unification of control- and data-flow problems, facilitating the computation of the actual (inter-procedural) flow of control as part of the DFA.

The design of the modified CBA framework, including lots of details and examples, can be found in (Jordan 2014, 198–289). Certain parts have been cherry-picked and summarised in the remainder of this section.

2.5.1 Property Space

A property space consists of a domain, representing all possible values a property may exhibit and a *combination operator*. The operator is used to merge the results of different control-flow paths considered by the analysis (Jordan 2014, 187). Using the reaching definitions example, presented previously, the property space would be given by

$$(2^D, \bigcup_\mathrm{D})$$

where

$$\begin{aligned} D &= V \times L \\ \bigcup_\mathrm{D} &: 2^D \to D \\ \{d_1, d_2, \dots, d_N\} &\mapsto \bigcup_{1 \leq i \leq N} d_i \end{aligned}$$

Each property space is inducing a lattice $(D, \subseteq)$ by defining $a \subseteq b \Longleftrightarrow b = \bigcup\{a, b\}$. Thus, $\bot = \bigcup\emptyset$ and $\top = \bigcup D$.

2.5.2 Analysis Variables

Analysis variables are given by a pair connecting the type of analysis (eg arithmetic value) and some id associated with an element of an underlying structure. For example, given the labelled expression $[[7]^1 + [4]^2]^3$ where the labels are written in superscript, the variable $A(3)$ refers to the *arithmetic value* of the expression labelled 3. This would lead to the following set of constraints.

$$\begin{aligned} \{7\} &\subseteq A(1) \\ \{4\} &\subseteq A(2) \\ \{x + y \mid x \in A(1),\ y \in A(2)\} &\subseteq A(3) \end{aligned}$$

2.5.3 Assignment

An assignment maps each analysis variable to a value of the corresponding property space. At the beginning of an analysis each analysis variable is initialised with the $\perp$ value of the respective property space. The solver (see below) modifies this mapping until all constraints are satisfied. The final assignment corresponds to the (least) solution of the system of constraints.

2.5.4 Constraint Guards

A detail omitted in the previous example is the option of having *conditional constraints*. These are, in other words, *guarded* constraints which are only considered if their *guard* evaluates to true. Taking one of the previous constraints we encountered out of context and attaching a random guard, just for the sake of it, gives us the following expression

$$x = 4 \land y < 2 \implies \mathrm{in}(4) \setminus \{(z,l) \mid l \in L\} \supseteq \mathrm{out}(4)$$

, where $x = 4 \land y < 2$ is the guard and $\mathrm{in}(4) \setminus \{(z,l) \mid l \in L\} \supseteq \mathrm{out}(4)$ is the constraint. Also, the term on the right of the $\supseteq$ operator must be a single variable – the constraint or target variable.

Constraints without an explicit guard can be viewed as one with a guard that is always true.

2.5.5 Constraint Generator

A constraint generator constructs new analysis variables and constraints. The constraints are attached to the variable and may depend on other analysis variables. Hence connections (ie dependencies) between variables are established. For new analyses, a constraint generator must be provided. It hooks into the framework and is used for the corresponding analysis. The framework already comes with some analyses and therefore their related property spaces and constraint generators. They can be used as a starting point for new analyses.

2.5.6 Analysis

An analysis combines a given property space with a constraint generator. It (and its variables) are distinguished from other analyses (and their variables) by an identifier. Already mentioned examples have been A, $\mathrm{REACH}^{\mathrm{in}}$, and $\mathrm{REACH}^{\mathrm{out}}$.

2.5.7 Constraint Solver

As presented by Nielson, Nielson, and Hankin (1999), the solver was required to have a *complete* set of constraints from the get go. The modifications made by Jordan (2014) include a *lazy-solver* capable of incorporating constraints during the solving process. Furthermore, this addition enables analyses to be more flexible since they can take temporary results into account – partial solutions obtained by the solver can influence the generation process of new constraints.

Additionally, in the original framework constraint guards were required to be *monotone* predicates. With the new *local restart* feature this restriction can be relaxed. This enables the analysis to make assumptions. This can lead to more accurate results, if the assumptions turn out to be valid. But results obtained from an assumption that turns out to be invalid need to be *forgotten* through resetting a set of interdependent analysis variables. This mechanism works fine as long as reset-events do not repeatedly trigger each other[2] (see Jordan 2014, 216).

[2] This problem does not occur if the constraint set is *stratifiable*.

The constraint solver itself is a fixpoint solver which utilises worklists. The algorithm initialises the assignment with the property spaces' $\perp$ values and updates it step by step until all constraints are satisfied. Solver covers its internal working.

3 Architecture

This chapter provides a grand overview of the system's architecture. It starts with a brief outline of the Insieme project and continues with investigating the INSieme Parallel Intermediate REpresentation (INSPIRE). Next, each module of the toolkit is described shortly by summarising its purpose.

3.1 Insieme

The Insieme project is split into two parts, the Insieme compiler and the Insieme Runtime. Both are designed in such a way that they can also be used separately. Figure 5 displays the typical pipeline where the compiler is combined with the runtime library. Insieme's mission statement follows (Insieme Team 2017).

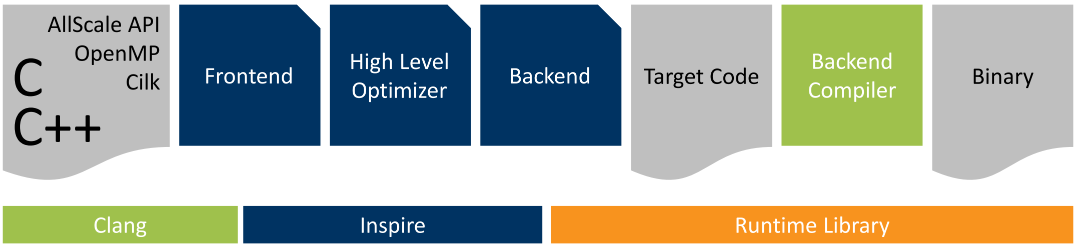


Figure 5: The Insieme compiler architecture.

> 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's Runtime maintains a set of worker threads to concurrently processing tasks. It knows about the system architecture and can take records of previously run programs for scheduling and optimisation decisions into account. We won't concern ourselves with the runtime any further since no interaction with it is required for this work.

Insieme's compiler is a source-to-source compiler taking C/C++ code as input and generating standard C++11 code as output (optionally with OpenCL. Along with this, the frontend accepts the AllScale API, OpenMP and Cilk for parallelism. An important aspect of Insieme is that it is built as a framework, not just a single compiler with a fixed set of options. This design decision keeps Insieme's architecture modular and prevents interference between modules.

Despite resembling a framework, the design is still similar to other compilers. It consists of a frontend, a backend and some complex parts in-between them. The frontend's purpose is to validate and transform input code into INSPIRE. The original input code is no longer needed after transformation. The INSPIRE program is all that is needed for subsequent steps and is capable of modelling parallel constructs and control-flow. The complex parts are made up of the core module (mainly responsible for managing INSPIRE), the analysis module and the transformation module. The analysis module is relevant for identifying code optimisation candidates and feature extraction. It takes a program represented in INSPIRE as input and, depending on the configuration, runs various analyses on it. The information derived in this step can further be used in the transformation module to realise optimisations. The optimised program can then be fed to the backend, which is responsible for producing target code.

3.2 INSPIRE

In order to use the Insieme infrastructure and create analyses with the new toolkit, a certain understanding of Insieme's Intermediate Representation (IR) is required. This section communicates the most important aspects of INSPIRE. The information provided here comes in handy, when implementing an analysis.

INSPIRE has multiple different representations. Inside the compiler it is handled as a Direct Acyclic Graph (DAG), but a text representation resembling a functional programming language is available as well, for development and testing purposes. *Pretty printer* and *parser* are part of the infrastructure allowing conversions between the DAG and text representation, while a binary dumper as well as a JSON dumper are provided for exporting INSPIRE. These exports can then be used by other tools like the Haskell-based Analysis Toolkit or Appendix A: INSPYER.

The structure of INSPIRE is presented next.

3.2.1 Structure

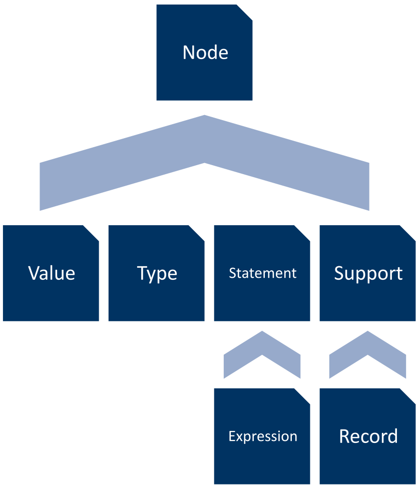


Figure 6: Hierarchy composed of node categories.

Each node belongs to a category which are hierarchically organised, as can be seen in Figure 6. On top we have the base class Node from which the six categories Value, Type, Statement, Program and Support are derived. Furthermore Expression is derived from Statement and Record is derived from Support.

In this subsection the *complete* structure of INSPIRE is given, yet only selected node types, relevant for the remainder of this document, are described. For further information see (Jordan et al. 2013) and (Jordan 2014, 37–172).

3.2.1.1 Notation

The block below is just an example used to communicate the notation used in this section. Categories are always written in italic, while the node type defined is written in bold. The types stated after a node definition are the fields (children) of that node. The following block defines the node categories *Example*. It contains four different types of nodes, BreakStmt, CompoundStmt, ForStmt and DeclarationStmt. BreakStmt has no children and is therefore a terminal. CompoundStmt contains a list (annotated by the square brackets) of *Statement*s. ForStmt holds a DeclarationStmt, two *Expression*s followed, by a CompoundStmt. Lastly, the DeclarationStmt consists of a Declaration and a Variable. Note that member field indices are currently the main method of addressing sub-structures in HAT.

Example ::= **BreakStmt**
| **CompoundStmt** [*Statement*]
| **ForStmt** DeclarationStmt *Expression* *Expression* CompoundStmt
| **DeclarationStmt** Declaration Variable

The actual structure of INSPIRE follows.

3.2.1.2 Nodes

Node ::= *Value*
| *Type*
| *Statement*
| *Support*
| **Program** [*Expression*]

This category builds the root of the hierarchy and only defines one node type itself. Program represents an entire program with its entry points.

3.2.1.3 Value Nodes

Value ::= **BoolValue** `bool`
| **CharValue** `char`
| **IntValue** `int`
| **UIntValue** `unsigned`
| **StringValue** `std::string`

These nodes are simple wrappers around values of the stated C++ type.

3.2.1.4 Type Nodes

Type ::= **FunctionType** Types *Type* UIntValue Types
| **GenericType** StringValue Parents Types
| **GenericTypeVariable** StringValue Types
| **NumericType** *Expression*
| **TagType** TagTypeReference TagTypeDefinition
| **TagTypeReference** StringValue
| **TupleType** [*Type*]
| **TypeVariable** StringValue
| **VariadicGenericTypeVariable** StringValue Types
| **VariadicTypeVariable** StringValue

GenericTypes and GenericTypeVariables are used for abstract operators (see Extension Mechanism) and, hence mainly for language extensions. TagTypes, on the other hand, are for records / structs. Yet, these nodes are of little relevance to the program analysis since the semantics are mainly modelled by the language extensions. Also note that the framework assumes a valid input program, therefore no type checking is required by the framework.

3.2.1.5 Statement Nodes

Statement ::= *Expression*
| **BreakStmt**
| **ContinueStmt**
| **GotoStmt** StringValue
| **LabelStmt** StringValue
| **CompoundStmt** [*Statement*]
| **DeclarationStmt** Declaration Variable
| **IfStmt** *Expression* CompoundStmt CompoundStmt
| **WhileStmt** *Expression* CompoundStmt
| **ForStmt** DeclarationStmt *Expression* *Expression* CompoundStmt
| **MarkerStmt** UIntValue *Statement*
| **ReturnStmt** Declaration
| **SwitchStmt** *Expression* SwitchCases CompoundStmt
| **ThrowStmt** *Expression*
| **TryCatchStmt** CompoundStmt [CatchClause]

The provided statements are equivalent to their C/C++ counterparts, just note the ForStmt which is actually a range-based for-loop. The Expression inside its Declaration marks the start, while the other two Expressions inside the ForStmt are the end and step, respectively. The MarkerStmt wraps another Statement connecting it to a unique identifier which can be used later on. This is especially helpful for testing and debugging purposes.

3.2.1.6 Expression Nodes

Expression ::= **BindExpr** FunctionType Parameters CallExpr
| **CallExpr** *Type* *Expression* [Declaration]
| **CastExpr** *Type* *Expression*
| **InitExpr** GenericType *Expression* Expressions
| **JobExpr** GenericType *Expression* *Expression*
| **LambdaExpr** FunctionType LambdaReference LambdaDefinition
| **LambdaReference** FunctionType StringValue
| **Literal** *Type* StringValue
| **MarkerExpr** *Type* UIntValue *Expression*
| **TupleExpr** TupleType Expressions
| **Variable** *Type* UIntValue

Like Statement nodes, Expression nodes are akin to C/C++. Similar to the statements, expressions can be marked too using MarkerExpr. BindExpr enables the construction of closures, while JobExpr is used for parallel constructs.

3.2.1.7 Support Nodes

Support ::= *Record*
| **CatchClause** Variable CompoundStmt
| **Declaration** *Type* *Expression*
| **Expressions** [*Expression*]
| **Field** StringValue *Type*
| **Fields** [Field]
| **Lambda** FunctionType Parameters CompoundStmt
| **LambdaBinding** LambdaReference Lambda
| **LambdaDefinition** [LambdaBinding]
| **MemberFunction** StringValue BoolValue *Expression*
| **MemberFunctions** [MemberFunction]
| **Parameters** [Variable]
| **Parent** BoolValue UIntValue *Type*
| **Parents** [Parent]
| **PureVirtualMemberFunction** StringValue FunctionType
| **PureVirtualMemberFunctions** [PureVirtualMemberFunction]
| **SwitchCase** Literal CompoundStmt
| **SwitchCases** [SwitchCase]
| **TagTypeBinding** TagTypeReference *Record*
| **TagTypeDefinition** [TagTypeBinding]
| **Types** [*Type*]

These nodes are utilised for building up types, expressions and statements. As the name of this node category hints, the purpose of these nodes is to keep the remaining structure of INSPIRE less complex and easier to work with.

3.2.1.8 Record Nodes

Record ::= **Struct** StringValue Fields Expressions Expressions BoolValue
MemberFunctions PureVirtualMemberFunctions Parents
| **Union** StringValue Fields Expressions Expressions BoolValue
MemberFunctions PureVirtualMemberFunctions

Struct is used to model C++ classes, including virtual functions and inheritance. The Parents node holds a list of parent (ie base) classes plus information about the type of inheritance (public, protected, or private). A Struct is composed of (in order): name, fields, constructors, destructors, destructor virtual flag, member functions, and pure virtual member functions. Since unions can not inherit structure, Union does not feature Parents.

3.3 Extension Mechanism

INSPIRE supports *language extensions*, similar to (abstract) libraries, which allow the structured integration of complex functionality (eg parallel directives). Insieme itself already uses this mechanism to model the semantics of C++ programs.

Language extensions do not add additional node types, instead they build *abstract data types* (and operators) by combining the previously (see INSPIRE) defined node types (core language). An abstract data type (not to be confused with Algebraic Data Types (ADTs)) is defined by its semantics and handled as a black box. For example, the abstract data type `int<4>` represents a 4 byte wide integer, yet we do not state *how* this integer is represented in its binary form.

Abstract operators enable us to work with abstract data types, despite not having their inner workings defined. For instance, the abstract operator `int_add` is defined semantically as adding two integers together, however we do not state how exactly this addition is to be performed.

By combining abstract operators we are able to craft new, *derived* operators having their semantics defined programmatically. For instance, given the abstract integer operation modulo (`%`) and equality (`==`), we are able to define a derived operator calculating the *greatest common divisor* of two integers. The definition combines aforementioned abstract operators with constructs of the INSPIRE core language (eg IfStmt, ReturnStmt, etc). In INSPIRE these derived operators are realised as lambdas.

Each abstract data type and operator is available in INSPIRE under its own name and operators are called like regular functions using a CallExpr. A more detailed description of the extension mechanism can be found in (Jordan 2014, 119–41).

Extensions are defined in the `lang` subdirectory of the Insieme core module via the use of preprocessor macros. The following list displays the different options available for building abstract data types and operators.

Abstract / Composed Type

A new, generic abstract data type can be created using the `LANG_EXT_TYPE` macro. The two arguments are the new abstract data type's handle in C++ and the *specification* in INSPIRE. The following code snippet creates the boolean and 4 byte wide integer types.

```c++
LANG_EXT_TYPE(Bool, "bool")
LANG_EXT_TYPE(Int4, "int<4>")
```

Composed types are created by combining type constructors, like tuple, struct, or function type with already declared data types. The following code snippet composes the type `int4_pair` using two 4 byte wide integers.

```c++
TYPE_ALIAS("int4_pair", "(int<4>, int<4>)")
```

Abstract / Derived Operators

An abstract operator is declared using the `LANG_EXT_LITERAL` macro with an id (C++ handle), name, and its respective type. The integer modulo operation and equality are provided as examples.

```c++
LANG_EXT_LITERAL(IntAdd, "int_mod", "(int<'a>, int<'a>) -> int<'a>")
LANG_EXT_LITERAL(IntEq,  "int_eq",  "(int<'a>, int<'a>) -> bool")
```

Derived operators are defined using `LANG_EXT_DERIVED`. The name, under which they are available in INSPIRE is derived from their C++ handle (first argument) by converting it from *CamelCase* to *snail_case* – in this case `int_gcd`.

```c++
LANG_EXT_DERIVED(IntGcd, "(a : int<'a>, b : int<'b>) -> int<'a> {"
"  if (int_eq(a, 0)) {"
"     return b;"
"  } else {"
"    return int_gcd(int_mod(b, a), a);"
"  }"
"}")
```

Constants

Constants are actually a special case of abstract operators. They are abstract operators which's type is not a function type. We therefore use the same macro as for creating abstract operators. Here the two boolean constants true and false are declared.

```c++
LANG_EXT_LITERAL(True, "true", "bool")
LANG_EXT_LITERAL(True, "false", "bool")
```

Constants are used directly in INSPIRE using a Literal node – no CallExpr is needed.

Operators and Analyses

The operators of language extensions represent semantics which need to be handled by an analysis. For abstract operators, the analysis designer *has to* provide an `OperatorHandler` (see Operator Handler) to interpret their semantics accordingly. This is optional for derived operators, as their semantics are encoded in their definitions.

3.3.1 The Basic Language Extension

The list of node types of INSPIRE does not contain primitive types and operations like integers or boolean, nor does it feature basic operators like `+` or `||`. These are defined in the *basic language extension* using a similar set of macros as previously introduced by this section.

The content of this language extension can further be divided into logical sections, which are presented next. The implementation of the content covered in this section can be found in:

`code/core/include/insieme/core/lang/inspire_api/basic.def`

3.3.1.1 Identifiers

The *basic language extension* defines identifiers as a distinct type of the INSPIRE language. There do not exist any operators for this type.

```C++
TYPE(Identifier, "identifier")
```

This abstract data type is used by the identifier analysis in Identifier Analysis to determine the value of expressions representing identifiers.

3.3.1.2 Arithmetic Types and Operators

This part of the basic language extension defines various numeric types and operations. Here we find the generic integer type`int<'a>` as well as its specialisations (eg `int<4>`, `int<16>`, etc). All primitive operations (eg `int_add`, `uint_lshift`, etc) are defined as abstract operators. The following table shows the five basic integer operations.

Abstract operators defined by the *basic language extension* for integer operations.

Operator	Type	Description
`int_add`	`(int<'a>,int<'a>) -> int<'a>`	Integer addition.
`int_sub`	`(int<'a>,int<'a>) -> int<'a>`	Integer subtraction.
`int_mul`	`(int<'a>,int<'a>) -> int<'a>`	Integer multiplication.
`int_div`	`(int<'a>,int<'a>) -> int<'a>`	Integer division.
`int_mod`	`(int<'a>,int<'a>) -> int<'a>`	Integer modulo.

The semantics implied by these operations needs to be handled by the corresponding analysis (in this case the arithmetic analysis, described in Arithmetic Analysis).

3.3.1.3 Boolean Types and Operations

The boolean type with the two literals `true` and `false` is defined as follows:

```C++
TYPE(Bool, "bool")
LITERAL(True,  "true",  "bool")
LITERAL(False, "false", "bool")
```

The three boolean operations *and*, *or*, and *not* are implemented as derived operators modelling short-circuit evaluation. They are summarised by the following table.

Derived operators defined by the for boolean operations.

Operator	Type	Description
`bool_not`	`(bool) -> bool`	Boolean *not.*
`bool_and`	`(bool,()=>bool) -> bool`	Boolean *and* supporting short-circuit evaluation.
`bool_or`	`(bool,()=>bool) -> bool`	Boolean *or* supporting short-circuit evaluation.

Numeric comparisons also fall in this part and are defined as abstract operators for the types`char`, `int<'a>`, `uint<'a>` and `real<'a>`. The following table summarises them for the type `int<'a>`.

Abstract operators defined by the *basic language extension* for integer comparison.

Operator	Type	Description
`int_eq`	`(int<'a>, int<'a>) -> bool`	Integer equality comparison.
`int_ne`	`(int<'a>, int<'a>) -> bool`	Integer inequality comparison.
`int_lt`	`(int<'a>, int<'a>) -> bool`	Integer less than comparison.
`int_gt`	`(int<'a>, int<'a>) -> bool`	Integer greater than comparison.
`int_le`	`(int<'a>, int<'a>) -> bool`	Integer less than or equal comparison.
`int_ge`	`(int<'a>, int<'a>) -> bool`	Integer greater than or equal comparison.

3.3.2 Reference Extension

In INSPIRE, references are the only means to address memory locations and memory locations are used for modelling mutual state.

Implicit and explicit C/C++ references are modelled by this extension. Implicit references are translated to *plain* references in INSPIRE. Explicit references, as available in C++, are mapped to `cpp_ref` or `cpp_rref` depending on whether we are dealing with an ordinary, l-value, or r-value reference. In addition to the reference kind, it can be stated if the referenced variable is `const`, `volatile`, or both.

Several operators are provided for memory management. For allocation the generic operator`ref_alloc` is provided returning a reference to a memory location allocated to fit an element of the requested type. This function can be combined with memory location information to specify whether the data is allocated on the heap or stack. `ref_delete` allows memory to be freed after it is no longer needed.

Accessing the data stored in a memory location addressed by a reference for reading is done by using`ref_deref` and `ref_assign` for writing. The following two tables provide an overview of common abstract (first) and derived (second) operators defined by this language extension.

Abstract operators defined by the reference language extension.

Operator	Type	Description
`ref_alloc`	`(type<'a>, memloc) -> ref<'a,f,f>`	Allocates memory for an object of given type at a given memory location.
`ref_decl`	`(type<ref<'a,'c,'v,'k>>) -> ref<'a,'c,'v,'k>`	References memory allocated in a surrounding declaration context.
`ref_delete`	`(ref<'a,f,'v>) -> unit`	Frees memory of the given reference.
`ref_deref`	`(ref<'a,'c,'v,'k>) -> 'a`	Used to obtain the data stored in the memory location linked to the given reference.
`ref_assign`	`(ref<'a,'c,'v,'k>, 'a) -> unit`	Used to update the value stored in the memory location linked to the given reference.
`ref_reinterpret`	`(ref<'a,'c,'v,'k>, type('b)) -> ref<'b,'c,'v,'k>`	A reinterpret cast altering the actual interpretation of the referenced memory cell.
`ref_narrow`	`(ref<'a,'c,'v,'k>, datapath<'a,'b>) -> ref<'b,'c,'v,'k>`	Obtain a reference to a sub-object within a referenced object.
`ref_expand`	`(ref<'b,'c,'v,'k>, datapath<'a,'b>) -> ref<'a,'c,'v,'k>`	The inverse operation to `ref_narrow`.
`ref_null`	`(type<'a>, type<'a>, type<'v>) -> ref<'a,'c,'v,plain>`	Creates a *null-reference* pointing to no memory location.
`ref_cast`	`(ref<'a,'c,'v,'k>, type<'nc>, type<'nv>, type<'nk>) -> ref<'a,'nc,'nv,'nk>`	A simple reference cast merely altering the view on the otherwise untouched memory location.

Derived operators defined by the reference language extension.

Operator	Type	Description
`ref_new`	`(type<'a>) -> ref<'a,f,f>`	Allocates memory for an object of given type on the heap by using `ref_alloc`.
`ref_array_element`	`(ref<array<'a,'s>,'c,'v,plain, int<8>) -> ref<'a,'c,'v,plain>`	Provides access to an element in an array by using`ref_narrow`.
`ref_member_access`	`(ref<'a,'c,'v,'k>, identifier, type<'b>) -> ref<'b,'c,'v,plain>`	Provides access to an element of a struct / union by using`ref_narrow`.
`ref_scalar_to_array`	`(ref<'a,'c,'v,plain>) -> ref<array<'a>,'c,'v,plain>`	A reference-navigation operator providing an array view on a scalar using `ref_expand`.

🗎 `code/core/include/insieme/core/lang/reference.h`

3.3.3 Parallel Extension

Up until this point we have not really talked about the parallel constructs available in INSPIRE. This extension covers the relevant primitives. At the most fine grained level INSPIRE provides parallelism by allowing multiple threads to be spawned at once, resulting in a *thread group* and joining them later on. The thread group is always tasked with a specific *job*. Basic identification queries like getting the index of the current thread in a group and getting the overall group size are available. Common parallel directives like barriers, mutexes and atomics are provided as well.

This extensions will by investigated further as it is not relevant to this work.

🗎 `code/core/include/insieme/core/lang/parallel.h`

3.4 Semantics

In this section we look at some example input codes, and compare them to the corresponding INSPIRE representations.

3.4.1 Empty Program

Below you can see the C++ input program on the left, the corresponding INSPIRE pretty printer output on the right and the simplified tree structure below. Omissions are indicated by an ellipsis.

```
// File: empty.cpp

int main() {
    return 0;
}
```

```
// File: empty.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main () {
  return 0;
}
```

```
// File: empty.tree

(Program |
    (LambdaExpr |
        (FunctionType | ... )
        (LambdaReference |
            (FunctionType | ...)
            (StringValue "IMP_main"))
        (LambdaDefinition |
            (LambdaBinding |
                (LambdaReference | ... )
                (Lambda |
                    (FunctionType | ... )
                    (Parameters )
                    (CompoundStmt |
                        (ReturnStmt | ... )))))))
```

The pretty printer output starts off with a declaration of the function `IMP_main`, here `IMP` stands for Insieme Mangling Prefix. It is followed by the definition of `IMP_main`. `() -> int<4>` is the type of `IMP_main` and states that no parameters are taken and an integer of 4 bytes is returned.

In the tree structure we see that the root node is a Program containing a LambdaExpr. Every function in the input code is translated to a lambda in INSPIRE (unless inlined). The lambda is composed of LambdaExpr, LambdaReference, LambdaDefinition, LambdaBinding and finally Lambda nodes. The Lambda node holds the CompoundStmt of the `main` function with its single ReturnStmt. Note that there are no Declarations in the actual IR structure. The Declaration seen in the pretty print is a presentation artefact. So are comments.

3.4.2 Variable Declaration

Next we define a single integer variable and observe the resulting INSPIRE output.

```
// File: decl.cpp

int main() {
    int i = 42;
    return 0;
}
```

```
// File: decl.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main (){
    var ref<int<4>,f,f,plain> v1 = 42;
    return 0;
}
```

```
// File: decl.tree

(Program | ...
    (CompoundStmt |
        (DeclarationStmt |
            (Declaration |
                (GenericType | ... )
                (Literal |
                    (GenericType | ... )
                    (StringValue "42")))
            (Variable |
                (GenericType | ... )
                (UIntValue 1)))
        (ReturnStmt | ... ))...)
```

The original variable `i` gets translated to `v_1` of type `ref<int<4>,f,f,plain>`, meaning it is a *plain*[3] reference to an integer. The reference is neither `const` nor `volatile`.

[3] A reference might either be `plain`, `cpp_ref`, or `cpp_rref` depending on its C++ semantic.

The DeclarationStmt creates an uninitialised memory location for a 4 byte integer on the stack. It is automatically destroyed when the reference goes out of scope. The value 42 is used to initialise the memory location created by the Declaration, which ends up being referenced by `v_1`. All variables are indexed by natural numbers. In our case it receives an id of 1.

3.4.3 Basic Arithmetic

Now we add some basic arithmetic by multiplying two integer variables.

```
// File: mul.cpp

int main() {
    int a = 2;
    int b = 3;
    int c = a * b;
    return 0;
}
```

```
// File: mul.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main (){
    var ref<int<4>,f,f,plain> v1 = 2;
    var ref<int<4>,f,f,plain> v2 = 3;
    var ref<int<4>,f,f,plain> v3 = (*v1) * (*v2);
    return 0;
}
```

```
 1  // File: mul.tree
 2
 3  (Program | ...
 4      (CompoundStmt |
 5          (DeclarationStmt | ... )
 6          (DeclarationStmt | ... )
 7          (DeclarationStmt |
 8              (Declaration |
 9                  (GenericType | ... )
10                  (CallExpr |
11                      (GenericType | ... )
12                      (Literal |
13                          (FunctionType | ... )
14                          (StringValue "int_mul"))
15                      (Declaration |
16                          (GenericType | ... )
17                          (CallExpr |
18                              (GenericType | ... )
19                              (Literal |
20                                  (FunctionType | ... )
21                                  (StringValue "ref_deref"))
22                              (Declaration |
23                                  (GenericType | ... )
24                                  (Variable |
25                                      (GenericType | ... )
26                                      (UIntValue 1)))))
27                      (Declaration |
28                          (GenericType | ... )
29                          (CallExpr |
30                              (GenericType | ... )
31                              (Literal |
32                                  (FunctionType | ... )
33                                  (StringValue "ref_deref"))
34                              (Declaration |
35                                  (GenericType | ... )
36                                  (Variable |
37                                      (GenericType | ... )
38                                      (UIntValue 2)))))))
39              (Variable |
40                  (GenericType | ... )
41                  (UIntValue 3)))
42          (ReturnStmt | ... ))...)
```

Again, the variables `a` and `b` are translated to `v_1` and `v_2` respectively, referencing memory locations storing their corresponding integer value. When multiplying the referenced value, we need to dereference first using the dereference operator `*`. In C/C++ this *variable-dereferencing* is implicit, while in INSPIRE those steps are explicit.

Multiplication and dereferencing are part of the arithmetic and reference extension. They are provided as operators and can be called via CallExpr.

In the CompoundStmt (line 4 in `mul.tree`) we see two DeclarationStmts on top. They contain the same structure as in the previous example. The third DeclarationStmt, on the other hand, holds two CallExpr (lines 17 and 29 in `mul.tree`) for dereferencing the variables `v_1` and `v_2` and another CallExpr (line 10 in `mul.tree`) for multiplying the two referenced integer values.

3.4.4 Function Call

Let us investigate how parameters and return values are passed. We therefore call a function `square` passing in an integer parameter and receive the squared value as return.

```
// File: call.cpp

int square(int x) {
    return x * x;
}

int main() {
    int i = square(3);
    return 0;
}
```

```
// File: call.ir

decl IMP_main : () -> int<4>;
decl IMP_square : (int<4>) -> int<4>;
def IMP_square = function (v1 : ref<int<4>,f,f,plain>)
                          -> int<4> {
    return (*v1) * (*v1);
};
// Inspire Program
int<4> function IMP_main (){
    var ref<int<4>,f,f,plain> v2 = IMP_square(3);
    return 0;
}
```

```
 1  // File: call.tree
 2  
 3  (Program | ...
 4      (CompoundStmt |
 5          (DeclarationStmt |
 6              (Declaration |
 7                  (GenericType | ... )
 8                  (CallExpr |
 9                      (GenericType | ... )
10                      (LambdaExpr |
11                          (FunctionType | ... )
12                          (LambdaReference | ... )
13                          (LambdaDefinition |
14                              (LambdaBinding |
15                                  (LambdaReference | ... )
16                                  (Lambda |
17                                      (FunctionType | ... )
18                                      (Parameters |
19                                          (Variable |
20                                              (GenericType | ... )
21                                              (UIntValue 1)))
22                                      (CompoundStmt |
23                                          (ReturnStmt |
24                                              (Declaration | ... )))))))
25                      (Declaration |
26                          (GenericType | ... )
27                          (Literal |
28                              (GenericType | ... )
29                              (StringValue "3")))))
30              (Variable |
31                  (GenericType | ... )
32                  (UIntValue 2)))
33          (ReturnStmt | ... ))...)
```

In addition to the IMP_main declaration we now also have a declaration of IMP_square receiving and returning an int<4>. In the function definition the variable v_1 references the passed-in value. Note that since C/C++ is implicitly allocating memory for parameters passed by value, the parameter type is actually ref<int<4>,f,f,plain> instead of int<4>.

In the Abstract Syntax Tree (AST), the CallExpr (line 8 in call.tree) directly contains the LambdaExpr (line 10 in call.tree) holding the translated IMP_square function plus the argument to the function, in this case the literal 3 (line 27 in call.tree). The list of Declarations (line 25 in call.tree) in CallExpr reflects the arguments of the call. The IMP_square function's body is composed of a CompoundStmt (line 22 in call.tree) with a single ReturnStmt (line 23 in call.tree). The squaring of values happens inside the Declaration (line 24 in call.tree) like shown in the previous example.

3.4.5 For Loop

This example is given to illustrate the use of ForStmt.

```
// File: for.cpp

int main() {
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += i;
    }
    return 0;
}
```

```
// File: for.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main (){
    var ref<int<4>,f,f,plain> v1 = 0;
    {
        for( int<4> v20 = 0 .. 10 : 1) {
            comp_assign_add(v1, v20);
        }
    }
    return 0;
}
```

```
 1  // File: for.tree
 2  
 3  (Program | ...
 4      (CompoundStmt |
 5          (DeclarationStmt |
 6              (Declaration | ... )
 7              (Variable |
 8                  (GenericType | ... )
 9                  (UIntValue 1)))
10          (CompoundStmt |
11              (ForStmt |
12                  (DeclarationStmt |
13                      (Declaration |
14                          (GenericType | ... )
15                          (Literal |
16                              (GenericType | ... )
17                              (StringValue "0")))
18                      (Variable |
19                          (GenericType | ... )
20                          (UIntValue 20)))
21                  (Literal |
22                      (GenericType | ... )
23                      (StringValue "10"))
24                  (Literal |
25                      (GenericType | ... )
26                      (StringValue "1"))
27                  (CompoundStmt |
28                      (CallExpr |
29                          (GenericType | ... )
30                          (LambdaExpr |
31                              (FunctionType | ... )
32                              (LambdaReference |
33                                  (FunctionType | ... )
34                                  (StringValue "comp_assign_add"))
35                              (LambdaDefinition | ... ))
36                          (Declaration |
37                              (GenericType | ... )
38                              (Variable |
39                                  (GenericType | ... )
40                                  (UIntValue 1)))
41                          (Declaration |
42                              (GenericType | ... )
43                              (Variable |
44                                  (GenericType | ... )
45                                  (UIntValue 20))))))
46          (ReturnStmt | ... ))...)
```

The ForStmt models a range-based for-loop with its beginning, end, and step size. The compound assignment operator `+=` inside the for-loop is translated to the `comp_assign_add` derived operator.

For simplicity the Insieme frontend is translating every for-loop into a while-loop in a first step, before attempting to restore for-loops qualifying as range-based loops.

When the original for-loop was translated to a WhileStmt, it was placed in a new CompoundStmt (line 10 in`for.tree`) to account for the scoping of the index variable. This CompoundStmt remains even after the WhileStmt has been converted to a ForStmt. The ForStmt consists of a DeclarationStmt (line 12 in `for.tree`) initialising the loop index, two literals (lines 21 and 24 in`for.tree`) for marking the end and stepsize, and finally a CompoundStmt for the translated body of the original for-loop (line 27 in `for.tree`. Here it contains the call to the`comp_assign_add` operator (line 28 in `for.tree`).

The derived operator `comp_assign_add` is a lambda, including an implementation defining its semantic by combining other primitives.

3.5 Haskell-based Analysis Toolkit

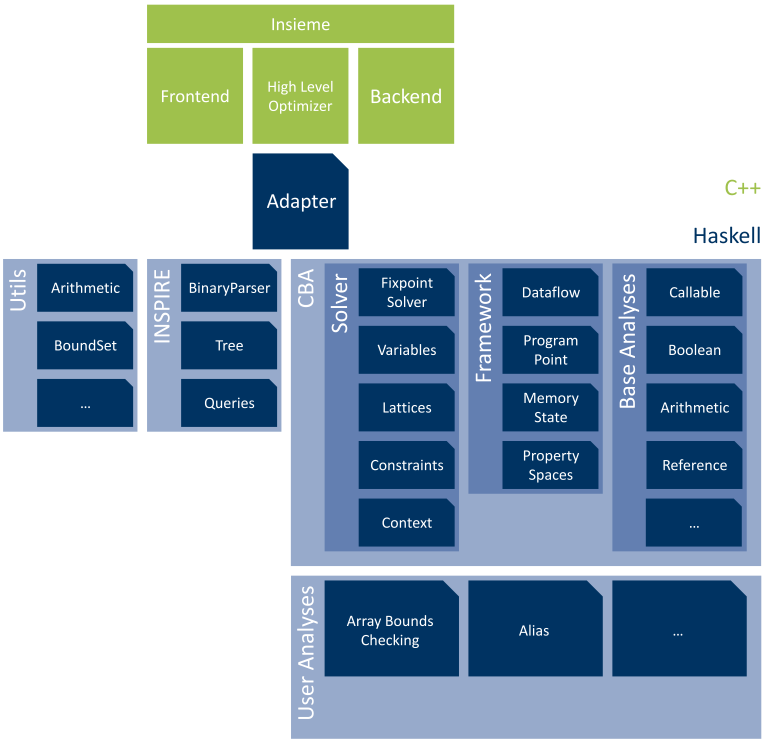


Figure 7: Simplified architecture of HAT, interfacing with Insieme.

In we see a slightly simplified design of HAT. At the top we can see Insieme with its basic structure and beneath it the Adapter module attaching the toolkit to the High Level Optimiser. The INSPIRE module is responsible for holding an input program and providing queries to specific parts of the underlying data structure. The analysis module, here titled CBA, can be divided into three submodules. Note that the architecture may evolve over time and certain parts of the module layout may be revisited in the future. This document describes the state of the toolkit at the time of writing (February 2017).

3.5.1 Adapter

The adapter is the only module of the toolkit existing in both, the C++ and Haskell domain of Insieme. It therefore bridges these two worlds via a *foreign function interface*. Data, relevant for the analysis, can be passed in both directions. Typically, an INSPIRE representation of an input program is transferred to the toolkit plus the information what analysis should be run on which part of the program. After the analysis has been completed the result may be transferred back to the C++ domain. Depending on the chosen analysis, the result can have many different formats (see analyses presented in Base Analyses).

3.5.2 Utilities

The two interesting submodules in Utils are Arithmetic and BoundSet. Arithmetic provides a data structure for simple arithmetic formulas, which can be combined using basic arithmetic operations. BoundSet is also a data structure, similar to the regular `Data.Set` found in Haskell's `containers` package[4]. The main difference is that you can set an upper limit on how many elements the set can hold. Upon adding another element, that is not already a member of the set, the set turns into `Universe` representing every possible value of the domain. This mechanism is required for some property spaces to ensure corresponding analyses to terminate.

[4] https://hackage.haskell.org/package/containers

3.5.3 INSPIRE

The input program, represented in INSPIRE, is transferred to the toolkit through a binary export. The INSPIRE module maintains a BinaryParser which takes the binary encoding as input and creates a representation of it, in the form of a *rose tree*, in Haskell. The related concepts of *node paths* and *node addresses* are translated as well and a collection of queries eases working with the tree.

A rose tree is defined as a tree where each node can have a variable number of children. The most basic definition looks like this (in Haskell):

```haskell
data Tree a = Node a [Tree a]
```

3.5.4 Framework

During an analysis the toolkit constructs additional data structures which expose critical information for analyses. These data structures include INSPIRE, Program Points, and Memory State information but can be extended as needed. The framework is responsible for elevating a user defined analysis to declarations, parameter and return value passing, closures, calls to unknown external functions, inter-procedural analyses and memory locations automatically. This is illustrated in Constructing a new Analysis and achieved via the use of base analyses.

3.5.5 Base Analyses

The generic Data-Flow Analysis, part of the framework, not only utilises multiple of these specialised analyses, but the Base Analysis themselves are specialisations of the generic DFA. Therefore the Base Analyses and Framework module depend on each other. These Base Analyses are necessary to correctly deduce the control flow of a program. For instance, the Arithmetic and Boolean value analysis are important for branches and loops, while Callable is vital for inter-procedural analyses.

3.5.6 User Analyses

Last but not least, user defined analyses build on top of the framework. While the illustration shows them combined into a single module, this is not necessary. How complex these analyses are is left to the user and we will see how Array Bounds Checking is realised later on in Constructing a new Analysis. Note that the framework imposes only few restrictions on what the property space can be. The downside of this feature is that the adapter needs to be extended in some cases. Otherwise the result cannot be used outside the Haskell domain.

4 Framework Implementation

Now, that our stage has been properly decorated with the required background knowledge and a grand overview, we can dive into the actual implementation of the framework.

First we take a look at the build infrastructure and the adapter used for communication between Haskell and C++. The modelling of the INSieme Parallel Intermediate REpresentation (INSPIRE) in Haskell follows. The fixpoint solver; together with modelling of arrays, structures, and unions; program points; and memory state is presented next. The final part investigates the base analyses required and provided by the framework.

4.1 Build Process

First, we examine the build process. Remember that the Adapter module exists in both, the Haskell and C/C++ domain, as it bridges the gap between them. Because of this, we do not solely rely on the Haskell Stack build tool but also on CMake, utilised by Insieme. For Insieme's fully automated build process, CMake invokes Haskell Stack with the required parameters to build the Haskell modules of the toolkit.

4.1.1 Compiling HAT into a Library

Haskell Stack is not only designed for automating the build process of Haskell packages, but also serves as a package manager to download, build and install third-party packages. Packages for this mechanism are hosted on Stackage in different versions. Furthermore Haskell Stack also sets up a suitable compiler for the chosen *release*. On Stackage a release, like `lts-7.0`, groups together a set of packages, a specific version for each of these packages, and a specific version of the Glasgow Haskell Compiler (GHC).

In our case we are using the release`lts-6.17` which brings `ghc-7.10.3` along. This release contains almost all packages we need, except `attoparsec-binary` and `directory`. Fortunately, Haskell Stack allows us to specify these requirements in the corresponding config file (`stack.yaml`) where we also state which release we are using. When specifying these extra dependencies, we also need to supply a specific version for each of them. In our case the entries are `attoparsec-binary-0.2` and `directory-1.2.6.3`.

Haskell Stack downloads all required packages and the Haskell compiler into a directory referred to as Stack Root. By default this is a folder named `.stack` inside the home-directory of the user invoking the Haskell Stack binary`stack`. This default setting can be overwritten by setting the `STACK_ROOT` environment variable when executing`stack`. The version of stack we are currently using (`1.0.4`) turned out to be quite picky about the ownership of the directory containing the Stack Root folder. If the directory belongs to a different user than the one invoking `stack`, it complains – even though the user has sufficient permissions to create the Stack Root folder.

A pre-compiled binary for Haskell Stack is available for common distributions and allows one to use it without having any Haskell related stuff installed. The only dependencies are the GMP and zlib libraries, which are widely available for any Linux based platform.

4.1.2 Linking the Shared Libraries

Stack itself is capable of building a shared library out of a Haskell project. Typically each third-party package is built as a shared library and so is the Haskell-based Analysis Toolkit (HAT). The library resulting from HAT, named `libHSinsieme-hat-0.2.so`, depends on the shared libraries of the third-party packages. Furthermore, neither the shared library of HAT nor any of the other third-parties contain the Haskell runtime environment. We therefore need to link the shared library `libHSrts-ghc7.10.3.so` containing the runtime explicitly when building Insieme.

During this process we discovered that Stack, at least in the version used, strips symbols, which are required for the linking process, from a shared library. It is therefore required to patch Haskell Stack, and hence remove this behaviour, since there was no option provided to prevent this from happening. The patch can be viewed in the following code snippet and shows that the only change needed is to pass the `--disable-library-stripping` flag to the Cabal library. The Cabal library is responsible for the compilation process and used by Stack. Luckily Insieme comes with a dependencies installer capable of building third-party software from source and injecting patches.

```
--- a/src/Stack/Types/Build.hs
+++ b/src/Stack/Types/Build.hs
@@ -606,6 +606,7 @@ configureOptsNoDir :: EnvConfig
                    -> [String]
 configureOptsNoDir econfig bco deps wanted isLocal package = concat
     [ depOptions
+    , ["--disable-library-stripping"]
     , ["--enable-library-profiling" | boptsLibProfile bopts || boptsExeProfile bopts]
     , ["--enable-executable-profiling" | boptsExeProfile bopts && isLocal]
     , ["--enable-split-objs" | boptsSplitObjs bopts]
```

A different issue, mainly arising from the way CMake operates, is *finding* the resulting shared library. Files generated during the build process, as well as the output files, are placed inside a folder named `.stack-work` inside the project directory. In our case the wanted shared library is located at

```
.stack-work/dist/x86\_64-linux/Cabal-1.22.5.0/build/libHSinsieme-hat-0.2-7YeplHX5wtl1qxdm3bLxLA-ghc7.10.3.so
```

Yet, the `.stack-work` directory does not exist prior to the build process and certain meta information is encoded in the path. Among architecture (`x86_64`), Cabal library version (`1.22.5.0`), and GHC version (`7.10.3`), a so-called *package key* (`7Yep1HX5wtl1qxdm3bLxLA`) is present. This package key is a hash of the package name, package version, and its dependencies. It is therefore very likely to change during development. CMake requires one to explicitly state the name of a shared library for linking plus its location, which must be known during the CMake run – before building the library. In our case, since the file name is likely to change, a workaround has been put into place. A script named `copyLibs.hs` is executed by CMake after invoking `stack`, which finds the shared library inside the `.stack-work` directory using a glob and copies it to CMake's build directory of HAT. Version and package key are stripped resulting in the file name `libHSinsieme-hat.so`.

Now linking this shared library can be stated explicitly in the CMake configuration, but the linker proofs to be to smart for our con. The original file name (containing version and package key) is present inside the shared library's meta data[5]. The linker now complains that it cannot find the file with this name – so we need another workaround for the workaround. This is achieved by copying the library to the build directory not only once, but twice. Once with the version and package key stripped from the file name so we can state it in the CMake configuration and once without anything stripped so the linker finds the correct file. This is to be found in the following file of the Insieme code base:

`code/analysis/src/cba/haskell/insieme-hat/CMakeLists.txt`

4.2 Adapter

The Adapter is responsible for attaching HAT to the Insieme compiler. It therefore consists of Haskell, as well as, C/C++ code. Through the use of a Foreign Function Interface (FFI) data can be transferred between the two domains. This works by exporting Haskell and C++ function in such a way that they can be linked and called like regular C functions. It follows that the names are not mangled and the platform's default C *calling convention* is used for these functions.

4.2.1 Foreign Function Interface

Both languages provide a simple mechanism to export functions for FFI. Two examples, one for C++ and one for Haskell are given.

C++

```
1  #include <iostream>
2  
3  extern "C" {
4      void hello_world(void) {
5          std::cout << "Hello World"
6                    << std::endl;
7      }
8  }
```

haskell

```
1  {-# LANGUAGE ForeignFunctionInterface #-}
2  
3  foreign export ccall "hello_world"
4    helloWorld :: IO ()
5  
6  helloWorld = putStrLn "Hello World"
```

In C++ the directive `extern "C"` is all that is required to export the function correctly. In the Haskell domain we first have to declare that we are using the FFI language extension in the first line. Next, in addition to a regular function definition (line 6) we declare that a function should be exported using C calling convention (`ccall`) and with the symbol `hello_world` in line 3. The following line (4) belongs to that FFI declaration and states which Haskell function is to be exported, including the type signature of that function. Note that all functions exported via the FFI live in the `IO` monad.

That alone is quite nice, but not very useful. We also require to*import* a function exposed via the FFI. The following code does exactly that.

C++

```
1  extern "C" {
2      void hello_world(void);
3  }
```

haskell

```
1  {-# LANGUAGE ForeignFunctionInterface #-}
2  
3  foreign import ccall "hello_world"
4    helloWorld :: IO ()
```

In C++ we just need to wrap the function declaration inside `extern "C"`. For Haskell instead of `export` we now use `import` followed by stating C calling convention and the name of the symbol to import. The following line (4) states under which name that function is made available in Haskell and which type signature it has.

While this example does not illustrate data being transferred between the two domains, it shows the basic structure of building an FFI. Primitives[6] can be passed back and forth just like regular function parameters / return values. Haskell's `Foreign.Ptr` and `Foreign.C.Types` modules contain the relevant types for sending and receiving data over the FFI. Additionally the module `Foreign.C.String` is dedicated to transferring C strings.

In this context we have two different types of pointers, depending on where the target pointed to is located. If the pointer refers to an object living in the C++ domain we use the generic type `Ptr a` in Haskell. When exposing a pointer to a Haskell object[7] we use `StablePtr a`. The function `newStablePtr :: a -> IO (StablePtr a)` gives us a pointer to a Haskell object and marks the object as *stable*, meaning it will not be moved or deallocated by the garbage collector. Similar to dynamic memory management, we later have to *un-stable* the object by calling `freeStablePtr`. Note that the pointers can be passed back and forth between the two domains as needed. To dereference a pointer in Haskell one can either use `deRefStablePtr` or `peek` depending on the type.

[5] see Executable and Linkable Format (ELF) which is used by the linker

[6] integers, doubles, and pointers

Most of the time we do not use `peek` directly. Since working with arrays and strings is quite common, Haskell serves us some convenience functions like `peekArray` and `withArrayLen` for reading, and `pokeArray` for writing.

Despite the fact that *marshalling* can be done in Haskell, we do not use it in the framework. INSPIRE, being the most complex data structure transferred from C++ to Haskell, is exported to its binary format, which is then parsed into a rose tree (see INSPIRE). To pass complex data structures from Haskell to C++, we first recreate the data structure in C++ and export its constructor(s) via FFI. An example of this is provided in the Arithmetic module (see [Arithmetic Formula]).

4.2.2 Haskell RunTime System

In order to use Haskell in a C++ project, like Insieme, we not only have to link together all the correct parts and use a correct FFI. We also have to initiate (and tear down) the Haskell RunTime System (Haskell RTS). The two functions `hs_init` and `hs_exit` fulfil this functionality, where the initialise function optionally takes arguments which are then passed to the Haskell RTS. Since GHC does no longer support calling `hs_exit` from a non-static object's destructor, the Haskell RTS is initialised and terminated by a static object. No arguments are passed to the runtime.

```C++
extern "C" {
    void hs_init(int*, const char**[]);
    void hs_exit(void);
}

class Runtime {
  public:
    Runtime() { hs_init(nullptr, nullptr); }
    ~Runtime() { hs_exit(); }
} rt;
```

`code/analysis/src/cba/haskell/runtime.cpp`

4.2.3 Context

For easier interaction between the two domains a *context* is established. This *context* consists of a Haskell object and a C++ object which are linked to each other by pointers. We will see its benefits when talking about the IR components.

On the Haskell end, the *context* is used to retain a received INSPIRE program and the solver's state. Holding on to the solver's state is important to reuse already computed information for new analysis runs. When using the toolkit without the Insieme compiler, this *context* can still be used by setting the reference to its corresponding C++ object to `nullPtr`. By allowing for this *dummy context* we do not require an additional interface for the toolkit because it is used without Insieme.

4.3 IR Components

Next, we take a look at the components relevant to model INSPIRE. The bottom layer is composed of the node types and a rose tree. Node addresses, visitors and queries build upon this. An INSPIRE program is converted to its binary representation (using Insieme's binary dumper) and passed to the toolkit via the FFI. The binary dump's format can be inferred manually from the Insieme source code[8]. Relevant parts are covered in the presentation of the binary parser.

The binary dump, representing an INSPIRE program also comes with a list that states operators used by that program together with a node address pointing to the function describing it.

4.3.1 Node Types

The full set of node types is defined in the `ir_nodes.def` file of the Insieme core. It is processed via the use of the C pre-processor to construct an enum holding all node types, which is then used by Insieme. Luckily we also have the option to process such a file in Haskell using `c2hs`. With this tool, we can get a full list of available node types in exactly the same order as the Insieme core does. Maintaining the order is important since the binary dump simply uses the index in this enum for communicating the node type. Yet we do not use this enum directly. What we seek is something like this:

```haskell
data NodeType = BoolValue   Bool
              | CharValue   Char
              | IntValue    Int
              | UIntValue   Int
              | StringValue String
              | CompoundStmt
              | IfStmt
              | WhileStmt
              | CompoundStmt
--            | ...
```

This data structure is a list of *all* possible node types, where the *value nodes* have their value already attached. With this approach we run into two problems. First, we cannot derive this exact type definition from `ir_nodes.def` using `c2hs`. Second, we cannot let the compiler derive an `enum` instance for this data structure. To circumvent these two problems we have to resort to Template Haskell (TH). TH offers a way to modify the Haskell syntax tree prior to compilation – it is basically a pre-process.

[7] By *Haskell object* in C++ we actually mean an instantiated Algebraic Data Type (ADT)

[8] `code/core/src/dump/binary_dump.cpp`

First the `ir_nodes.def` file gets included filling a C enum that is then converted to Haskell using `c2hs`. All entries of this enum are prefixed with `NT_`. Based on this enum TH is used to define a data structure `NodeType` similar to the enum and two functions `fromNodeType` and `toNodeType`. `NodeType` is composed of all node types (without the `NT_` prefix) and value nodes have their corresponding value attached to them. This is exactly the data structure we want.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Inspire/NodeType.chs`

4.3.2 Node Path

```haskell
type NodePath = [Int]
```

A *node path* is a list of child indices describing a path along the tree starting at some node. It is commonly used to reference a node in INSPIRE. Each index tells you which child to go to next. You have reached the target upon depleting the list.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Inspire/NodeAddress.hs`

4.3.3 Binary Parser

The binary parser of the Inspire module builds upon `Attoparsec`, which is a library for constructing combinator parsers. Together with the `attoparsec-binary` package it becomes quite easy to build a binary parser for the INSPIRE export. The binary dump is mainly composed of three differently sized integer types and strings. We therefore define ourselves the following parser combinators.

```haskell
anyInt8 :: Parser Int
anyInt8 = fromIntegral <$> anyWord8

anyInt16 :: Parser Int
anyInt16 = fromIntegral <$> anyWord16le

anyInt32 :: Parser Int
anyInt32 = fromIntegral <$> anyWord32le

parseString :: Parser String
parseString = liftM BS8.unpack $ take =<< anyInt32
```

Integers larger than one byte are dumped in little endian, while strings are always prefixed with their length and do not contain a terminator.

The header of the binary dump starts with the magic number `0x494e5350495245` and a list of *converters*. Like strings, lists are prefixed with the number of elements. Converters are only relevant when using annotations attached to the INSPIRE program and since we do not use them, they are discarded by the main parser. The header is parsed with this set of combinators.

```haskell
parseHeader :: Parser [String]
parseHeader = parseMagicNr *> parseConverters

parseMagicNr :: Parser Word64
parseMagicNr =  word64le 0x494e5350495245
            <|> fail "wrong magic number"

parseConverter :: Parser String
parseConverter = parseString

parseConverters :: Parser [String]
parseConverters = do
    n <- anyInt32
    count n parseConverter
```

The main part of the binary dump – the encoding of the IR nodes – is a list of nodes, where each node starts with an integer encoding its type. If the node is a value node, the value follows, otherwise a list of integers follows telling us the node's children. The integers are list indices referring to other nodes in the same list. Each node can be followed by a list of annotations which is discarded by our parser. We first parse the whole list of nodes from the dump to create an integer map of, so called, `DumpNode`s, where the key is the corresponding index in the list. `DumpNode` is a node where child nodes are referenced through indices, instead of actual nodes. The following code parses a single `DumpNode`.

```haskell
data DumpNode = DumpNode IR.NodeType [Int]

parseDumpNode :: Parser DumpNode
parseDumpNode = do
    -- get node type
    t <- toEnum <$> anyInt16

    -- create corresponding node
    n <- case t of
        -- value nodes
        IR.NT_BoolValue   -> DumpNode <$> IR.BoolValue <$> (/=0) <$> anyInt8     <*> pure []
        IR.NT_CharValue   -> DumpNode <$> IR.CharValue <$> chr   <$> anyInt8     <*> pure []
        IR.NT_IntValue    -> DumpNode <$> IR.IntValue            <$> anyInt32    <*> pure []
        IR.NT_UIntValue   -> DumpNode <$> IR.UIntValue           <$> anyInt32    <*> pure []
        IR.NT_StringValue -> DumpNode <$> IR.StringValue         <$> parseString <*> pure []

        -- other nodes
        _              -> do
            c  <- anyInt32
            is <- count c anyInt32
            return $ DumpNode (IR.fromNodeType t) (fromIntegral <$> is)

    -- skip annotations
    a <- anyInt32
    count a anyWord64le

    return $ n
```

After this phase is completed, the DumpNodes are gradually converted into a single rose tree. The first node, the one with index zero, marks the root of the tree. Nodes may be referenced by the indices multiple times, yet it is important to construct a node only once and reuse already constructed ones. Otherwise we end up with multiple instances of the same node in memory, wasting not only a big amount of space, but also a lot of time when checking for equality later on. This concept is referred to as *node sharing* and, while Haskell does not offer us a way to handle references explicitly, we can deal with them by carefully using the constructor and already defined nodes. Each constructed node is temporarily stored using the State monad and a map during the construction of the rose tree.

```haskell
connectDumpNodes :: IntMap.IntMap DumpNode -> IR.Tree
connectDumpNodes dumpNodes = evalState (go 0) IntMap.empty
  where
    go :: Int -> State (IntMap.IntMap IR.Tree) IR.Tree
    go index = do
        memo <- get
        case IntMap.lookup index memo of
            Just n  -> return n
            Nothing -> do
                let (DumpNode irnode is) = dumpNodes IntMap.! index
                children <- mapM go is
                let n = IR.mkNode index irnode children []
                modify (IntMap.insert index n)
                return n
```

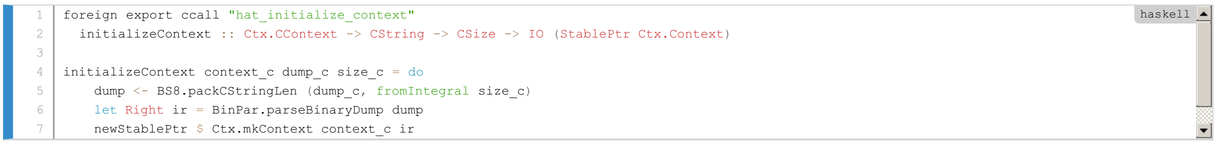
Now that we have the rose tree representing the received INSPIRE program we still need to do one slight modification. We go over each node one more time and attach information about operators (ie language extensions). This step is done by the function markBuiltins in line 15 (next code snippet). As already mentioned, after the binary dump of the INSPIRE program, a list of node paths addressing operators is provided by the binary dump.

We now have all the parts to construct the complete binary parser. The function parseBinaryDump combines what we have established in this section. It takes the whole binary dump as input and returns either the constructed rose tree or an error message.

```haskell
 1  parseBinaryDump :: BS.ByteString -> Either String IR.Tree
 2  parseBinaryDump = parseOnly $ do
 3      -- parse components
 4      parseHeader
 5      n            <- anyInt32
 6      dumpNodes    <- IntMap.fromList <$> zip [0..] <$> count n parseDumpNode
 7      m            <- anyInt32
 8      dumpBuiltins <- Map.fromList <$> count m parseBuiltin
 9      l            <- anyInt32
10      paths        <- count l parseNodePath
11
12      -- connect components
13      let root     = connectDumpNodes dumpNodes
14      let builtins = resolve root <$> dumpBuiltins
15      let ir       = markBuiltins root builtins
16
17      return ir
18
19    where
20        resolve :: IR.Tree -> [Int] -> IR.Tree
21        resolve node []     = node
22        resolve node (x:xs) = resolve (IR.getChildren node !! x) xs
```

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Inspire/BinaryParser.hs

Let us now take a final look at the adapter to see how this parser is used.



```haskell
foreign export ccall "hat_initialize_context"
  initializeContext :: Ctx.CContext -> CString -> CSize -> IO (StablePtr Ctx.Context)

initializeContext context_c dump_c size_c = do
    dump <- BS8.packCStringLen (dump_c, fromIntegral size_c)
    let Right ir = BinPar.parseBinaryDump dump
    newStablePtr $ Ctx.mkContext context_c ir
```

Upon setting up a context, the Haskell side receives a reference to the corresponding C++ object (`Ctx.CContext`) and the binary dump in form of a byte array (plus its length). This array is *packed* – necessary so the parser can work with it – and then parsed. The result is our rose tree. This allows us to setup an analysis context in Haskell.

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Adapter.hs`

4.3.4 Visitors

Since we have defined our own tree structure for INSPIRE we also have to provide some basic visitor functionality. For performance critical reasons one can stop the visiting process at a controlled depth. This is referred to as *pruning*. The following visitors are provided.

`foldTree`
: Folds the whole tree. No pruning takes place. `fold` is the Haskell equivalent of a reduction.

`foldTreePrune`
: Same as `foldTree`, but allows you to prune at any point.

`foldAddress`
: Folds the subtree starting at a given address. No pruning takes place.

`foldAddressPrune`
: Same as `foldAddress`, but allows you to prune at any point.

`collectAll`
: Returns a list of node addresses to all nodes satisfying a given predicate of the`NodeType`. No pruning takes place.

`collectAllPrune`
: Same as `collectAll`, but allows you to prune at any point.

`foldTree` and `foldTreePrune` operate on nodes while `foldAddress` and `foldAddressPrune` operate on node addresses.

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Inspire/Visit.hs`

4.3.5 Node References and Queries

To facilitate using the tree IR structure and node addresses a list of useful queries are implemented. These include, for example, checking if the given node is an operator or getting its type. The type-class `NodeReference` has been defined in order to allow nodes of the tree and node addresses to use the same queries, hence both of them implement this type class. The list of queries can be extended later on. Note that there exist a few[9] queries which can only be used with node addresses. This is due to the fact that a node address allows you to look at the parent of the referred node, which is not possible by using only plain nodes.

Furthermore a function `findDecl` is provided, which takes a node address pointing to a variable and tries to find its declaration.

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Inspire/Query.hs`

4.3.6 Context

Up until now we have already heard that a context is established in both domains and keeps track of the solver's state. We now take a look at the context's definition in C++ and how it is setup and used. The important parts are displayed below.

[9] Currently `isLoopIterator`, `hasEnclosingStatement`, `isEntryPoint`, and `isEntryPointParameter`.

```cpp
typedef void* StablePtr;
typedef void* HaskellNodeAddress;

class Context {

    StablePtr context_hs;

    core::NodePtr root;

    std::map<core::NodeAddress, HaskellNodeAddress> addresses;

  public:

    Context();
    explicit Context(const core::NodePtr& root);
    ~Context();

    // move semantics
    Context(const Context& other) = delete;
    Context(Context&& other) = default;

    Context& operator=(const Context& other) = delete;
    Context& operator=(Context&& other) = default;

    // -- haskell engine specific requirements --

    StablePtr getHaskellContext() const;
    void setHaskellContext(StablePtr);

    void setRoot(const core::NodePtr&);
    core::NodePtr getRoot() const;

    HaskellNodeAddress resolveNodeAddress(const core::NodeAddress& addr);
    core::NodeAddress resolveNodeAddress(const HaskellNodeAddress& addr);

};
```

The context holds a stable pointer to its counter part in the Haskell domain and a specific INSPIRE program is associated with it. Furthermore it caches already transferred node addresses. Since the context in C++ is responsible for freeing (ie *un-stabling* Haskell objects) it has a custom destructor and no copy semantics. Node addresses can be resolved in both directions using `resolveNodeAddress`. To use it, just create an instance of the class, passing the root of an INSPIRE program to the constructor. Functions initiating an analysis take the context as their first argument, which is then used to convert IR information. This allows us to isolate the FFI code from the rest of Insieme. Two function declarations, providing the user-interface to HAT-based analyses, are given as examples. Their signatures do not contain any Haskell related parts.

```cpp
bool mayBeTrue(Context& ctxt, const ExpressionAddress& expr);

ArithmeticSet getArithmeticValue(Context& ctxt, const ExpressionAddress& expr);
```

First the context needs to be established as described. Afterwards both functions can be used directly. What exactly these two functions do is covered later in the Base Analyses.

`code/analysis/include/insieme/analysis/cba/haskell/context.h`

4.4 Utilities

Before continuing with investigating the actual framework we take a look at some utilities used by the framework. The two most notable are bound sets and arithmetic formulas.

4.4.1 Bound Sets

A *bound set* is similar to a regular set, apart from the fact that it is associated with an upper bound. When the number of elements within the set exceeds this upper bound, the set turns into *universe* representing every possible value of the domain. Of course, using this set as property spaces for program analyses likely yields over approximations as soon as the set turns into *universe*. Yet it is necessary to maintain termination of the analyses (thus decidability). Note that the bound can be adjusted even during an analysis run to maximise flexibility, but adjusting it changes the type of the bound set. Its definition:

```haskell
data BoundSet bb a = BoundSet (Data.Set.Set a) | Universe
```

The functions provided are quite similar to the ones from `Data.Set`, as can be seen by the export list.

```haskell
module Insieme.Utils.BoundSet (
    BoundSet(Universe),
    empty, singleton,
    size, null, isUniverse, member,
    fromSet, fromList, toSet, toList,
    insert, applyOrDefault, filter, map, lift2,
    union, intersection, cartProduct,
    -- ...
) where
```

Bounds are defined in the following way.

```haskell
class Typeable b => IsBound b where
    bound :: p b a -> Int

getBound :: IsBound bb => BoundSet bb a -> Int
getBound bs = bound bs

changeBound :: (IsBound bb, IsBound bb', Ord a)
            => BoundSet bb a -> BoundSet bb' a
changeBound  Universe    = Universe
changeBound (BoundSet s) = fromSet s

data Bound10 = Bound10
instance IsBound Bound10 where
    bound _ = 10

data Bound100 = Bound100
instance IsBound Bound100 where
    bound _ = 100
```

The type-class requires the implementation of `bound` which takes a *bounded object* as input and returns the *capacity* associated with this bound. Two bounds, `Bound10` and `Bound100`, are already defined. More bounds can be added as needed. Having the bound encoded into the type of the bounded container prevents accidental combining of differently bound containers. A function `changeBound` can be used to explicitly convert between them.

Although we took `Data.Set` as a reference, bound sets are not interoperable with it. Yet, sometimes an *unbound* set is required (or helpful) and it would be convenient to use the same interface. We therefore define ourselves the `Unbound` bound.

```haskell
data Unbound = Unbound

instance IsBound Unbound where
    bound _ = -1

type UnboundSet a = BoundSet Unbound a

toUnboundSet :: (IsBound bb, Ord a) => BoundSet bb a -> UnboundSet a
toUnboundSet = changeBound
```

Some analyses yield a result in the form of a bound set (eg arithmetic value). Therefore a similar data structure exists at the C++ side. With it, results wrapped in a bound set can be transferred across the two domains as long as the element type is transferable.

Along with this, convenience functions are provided, among them `map`, `lift2`, `cartProduct`, and `applyOrDefault`.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Utils/BoundSet.hs`

4.4.2 Arithmetic Formulas

The purpose of the Arithmetic module is to model arithmetic formulas. For now only basic integer operations are implemented. On both sides – C++ and Haskell – a data structure is defined to model these formulas. Here is the definition of the Haskell part.

```haskell
-- v...variable type
-- c...constant type

-- Something like $x^2$.
data Factor c v  = Factor  { base     :: v
                           , exponent :: c }

-- Something like $x^2 y^2$.
data Product c v = Product { factors  :: [Factor c v] }

-- Somethign like $-2 x^2 y^3$.
data Term c v    = Term    { coeff    :: c
                           , product  :: Product c v }

-- Something like $-2 x^2 y^3 + 5 z^3 + 2$.
data Formula c v = Formula { terms    :: [Term c v] }
```

The types of variables `v` and coefficient / exponent `c` are generic. Node addresses are commonly used as variables to reference a variable (or expression) in the INSPIRE program. Using integer types from `Foreign.C.Types` instead of `Integer` allows more accurate modelling of overflow behaviour.

Two formulas can be added, subtracted and multiplied. Division by a constant is supported, but may only be accurate if the remainder is zero. For now this model seemed adequate, but can be replaced in the future by one more accurate.

In addition to the representation of arithmetic formulas, we bring a different definition of *numeric ordering* along.

```haskell
data NumOrdering = NumLT | NumEQ | NumGT | Sometimes

class NumOrd a where
    numCompare :: a -> a -> NumOrdering

fromOrdering :: Ordering -> NumOrdering
fromOrdering LT = NumLT
fromOrdering EQ = NumEQ
fromOrdering GT = NumGT
```

A formula can either always be greater than, less than, or equal to another formula; or the ordering depends on the value of variables (`sometimes`). These four different outcomes are modelled here.

For transferring an arithmetic formula from Haskell to C++ we have to reconstruct the formula part by part. For this purpose the corresponding constructors are wrapped and made available to the Adapter via the FFI as seen below.

```haskell
foreign import ccall "hat_mk_arithmetic_formula"
  arithmeticFormula :: Ptr (Ptr CArithmeticTerm) -> CSize -> IO (Ptr CArithmeticFormula)

foreign import ccall "hat_mk_arithmetic_product"
  arithmeticProduct :: Ptr (Ptr CArithmeticFactor) -> CSize -> IO (Ptr CArithmeticProduct)

foreign import ccall "hat_mk_arithmetic_term"
  arithmeticTerm :: Ptr CArithmeticProduct -> CULong -> IO (Ptr CArithmeticTerm)

foreign import ccall "hat_mk_arithemtic_factor"
  arithemticFactor :: Ptr CArithmeticValue -> CInt -> IO (Ptr CArithmeticFactor)

foreign import ccall "hat_mk_arithmetic_value"
  arithmeticValue :: Ctx.CContext -> Ptr CSize -> CSize -> IO (Ptr CArithmeticValue)
```

The following function does all the heavy lifting on the Haskell side.

```haskell
passFormula :: Integral c
            => Ctx.CContext
            -> Ar.Formula c Addr.NodeAddress
            -> IO (Ptr CArithmeticFormula)

passFormula ctx_c formula_hs = do
    terms_c <- forM (Ar.terms formula_hs) passTerm
    withArrayLen' terms_c arithmeticFormula
  where
    passTerm :: Integral c => Ar.Term c Addr.NodeAddress -> IO (Ptr CArithmeticTerm)
    passTerm term_hs = do
        product_c <- passProduct (Ar.product term_hs)
        arithmeticTerm product_c (fromIntegral $ Ar.coeff term_hs)

    passProduct :: Integral c => Ar.Product c Addr.NodeAddress -> IO (Ptr CArithmeticProduct)
    passProduct product_hs = do
        factors_c <- forM (Ar.factors product_hs) passFactor
        withArrayLen' factors_c arithmeticProduct

    passFactor :: Integral c => Ar.Factor c Addr.NodeAddress -> IO (Ptr CArithmeticFactor)
    passFactor factor_hs = do
        value_c <- passValue (Ar.base factor_hs)
        arithemticFactor value_c (fromIntegral $ Ar.exponent factor_hs)

    passValue :: Addr.NodeAddress -> IO (Ptr CArithmeticValue)
    passValue addr_hs = withArrayLen' (fromIntegral <$> Addr.getAbsolutePath addr_hs)
                                      (arithmeticValue ctx_c)

    withArrayLen' :: Storable a => [a] -> (Ptr a -> CSize -> IO b) -> IO b
    withArrayLen' xs f = withArrayLen xs (\s a -> f a (fromIntegral s))
```

The formula is reconstructed from the bottom up, where `value` here refers to node addresses and are the variables of the formula.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Utils/Arithmetic.hs`

4.5 Fixpoint Solver

This section describes the solver itself together with the `Lattice` type-class and used data structures for `Variable`s, `Assignment`s, and `Constraint`s.

4.5.1 Lattice

For implementation, two different names for the (complete) lattice and join-semi-lattice are used – `ExtLattice` and `Lattice` respectively.

```haskell
class (Eq v, Show v, Typeable v, NFData v) => Lattice v where
    join :: [v] -> v
    join [] = bot
    join xs = foldr1 merge xs

    merge :: v -> v -> v
    merge a b = join [a,b]

    bot :: v
    bot = join []

    less :: v -> v -> Bool
    less a b = (a `merge` b) == b

class (Lattice v) => ExtLattice v where
    top :: v
```

`join` describes how a list of values is combined into a single one. `merge` is just the binary version of `join` as can be seen by the default implementation. `less` determines whether one element of the lattice is less[10] than another element. This type-class must be instantiated by the property space of each analyses.

[10] $a < b \iff a$ is a more accurate value for an analyses than b

4.5.2 Variables

Analysis variables are described briefly in Analysis Variables. An analysis variable is associated with a list of constraints restricting potential `Assignment`s and a unique identifier. The definition follows.

```haskell
data Identifier = Identifier { analysis :: AnalysisIdentifier
                             , idValue  :: IdentifierValue
                             , idHash   :: Int }

data AnalysisIdentifier = AnalysisIdentifier { aidToken :: TypeRep
                                             , aidName  :: String
                                             , aidHash  :: Int }

data IdentifierValue = IDV_Expression NodeAddress
                     | IDV_ProgramPoint ProgramPoint
                     | IDV_MemoryStatePoint MemoryStatePoint
                     | IDV_Other BS.ByteString
  deriving (Eq,Ord,Show)


data Var = Var { index :: Identifier
               , constraints :: [Constraint]
               , bottom :: Dynamic
               , valuePrint :: Assignment -> String }

instance Eq Var where
        (==) a b = (index a) == (index b)

instance Ord Var where
        compare a b = compare (index a) (index b)


newtype TypedVar a = TypedVar Var
        deriving (Show,Eq,Ord)

mkVariable :: (Lattice a) => Identifier -> [Constraint] -> a -> TypedVar a
```

Note that most of the complexity of these definitions comes from the necessary performance optimisations – in particular for`Variable` comparison.

The identifier used for analysis variables contains an identifier for the corresponding type of analysis (`AnalysisIdentifier`) and a *value* (`NodeAddress`, `ProgramPoint`, `MemoryStatePoint`, or byte string) followed by a hash. The `Identifier`'s hash is used to speed up equality checks and takes the hash of `AnalysisIdentifier` into account.

The bottom value of an analysis variable depends on the analysis' property space. We therefore use `Dynamic` as type. We could use a generic type variable for this, but this would make it difficult to directly work with and combine variables of different analyses. See the related Haskell Wiki page for more information about using heterogeneous collections.

The structure `TypedVar` is used to statically connect a variable with its related property space – where possible. This is used in the `Assignment`. A `Var` is created and wrapped by `mkVariable` given an `Identifier`, a list of `Constraints`, and the bottom value of the corresponding property space. Only `TypedVar`s may be created.

4.5.3 Assignments

An `Assignment` is defined as a mapping from analysis variables to values of the related property space. The interface definition is given below.

```haskell
newtype Assignment = Assignment -- ...

get :: (Typeable a) => Assignment -> TypedVar a -> a

set :: (Typeable a, NFData a) => Assignment -> TypedVar a -> a -> Assignment
```

`get` returns the value associated to the given`TypedVar` in the `Assignment`. The use of `TypedVar` allows us to return the correct type of the value. `set` modifies an `Assignment` by associating a given `TypedVar` with the given value.

4.5.4 Constraints

```haskell
data Event = None
           | Increment
           | Reset

data Constraint = Constraint
    { dependingOn        :: Assignment -> [Var]
    , update             :: (Assignment -> (Assignment,Event))
    , updateWithoutReset :: (Assignment -> (Assignment,Event))
    , target             :: Var }
```

A `Constraint` is always associated with the one variable which's value it is constraining, here referred to as`target`. Depending on an `Assignment` a list of variables can be deduced on which the `Constraint` depends on.

A `Constraint` provides us with the two functions `update` and `updateWithoutReset` which mutate a given `Assignment`. The `update` function also gives us information about what happened in the form of an `Event`. The three possible outcomes are.

`None`

The `Assignment` did not change, we can continue without any additional work.

`Increment`

The `Assignment` changed during the update monotonously, all depending variables need to be reconsidered.

`Reset`

The current `Assignment` changed in a non-monotonous way, hence a *local restart* needs to be triggered.

The function `updateWithoutReset` is used to prevent local restarts from triggering each other by forcing monotonous updates. How exactly these three events are handled can be seen in Solver.

4.5.4.1 Dependencies between Variables

It is essential that the dependencies between variables is correctly established for all analyses. The framework's functions expect this to be done correctly be the analysis designer. This means that for a given `Assignment`, all analysis variables used to calculate the value of the current analysis variable must be present in the dependency list of the current variable. The correct use is illustrated in the following code snippet, given two analysis variables var_1 and var_2, and an `Identifier` for the new analysis variable (lines 1–3).

```haskell
 1  var1  :: Solver.TypedVar Int
 2  var2  :: Solver.TypedVar Int
 3  varId :: Identifier
 4
 5  con :: Constraint
 6  con = createConstraint dep val var
 7
 8  var :: Solver.TypedVar Int
 9  var = mkVariable varId [con] Solver.bot
10
11  dep :: Assignment -> [Var]
12  dep _ = [Solver.toVar var1, Solver.toVar var2]
13
14  val :: Assignment -> Int
15  val a = Solver.get a var1 + Solver.get a var2
```

In lines 6 and 9, a constraint and analysis variable are generated. Since the value (`val`) of the constraint depends on the values yielded by var_1 and var_2, the new analysis variable var depends on var_1 and var_2. This is stated in lines 6 and 12 where `dep` contains all analysis variable var is depending on, which is used to create the constraint.

4.5.5 Solver

Before talking about the solving process itself, we model the state of the solver. `SolverState`, as seen below, captures the current `Assignment` and all processed analysis variables (and thus their constraints) using the already introduced `VariableIndex` structure (see Variables).

```haskell
SolverState = SolverState { assignment    :: Assignment
                          , variableIndex :: VariableIndex }
```

The `solve` function takes an initial `SolverState` and a list of desired analysis variables as input, yielding a new `SolverState` containing the modified `Assignment`, containing the least fixpoint solution for the passed analysis variables.

```haskell
    solve :: SolverState -> [Var] -> SolverState
```

Behind the scene `solve` calls `solveStep` which contains the main solver logic.

```haskell
solveStep :: SolverState -> Dependencies -> [IndexedVar] -> SolverState
```

`solveStep` takes as input a `SolverState`, the `Dependencies` between variables and a *worklist* in form of a list of `IndexedVar`. The result is a new `SolverState`. The function takes one `IndexedVar` after another from the worklist (third parameter) and processes the constraints associated with this variable.

To process a constraint, the `update` function of this constraint is called with the current assignment, yielding a new assignment and an `Event` (see Constraints). In case the `Event` is `None`, the processing simply continues since the `Assignment` did not change. If it is `Increment` the constraints depending on the target variable of the current constraint need to be re-evaluated and the related variables are therefore added to the worklist. The old `Assignment` has been updated by the constraint's `update`. On `Reset` the function behaves similarly as on `Increment`, however the transitive closure of all dependent variables is reset to their respective bottom values – unless the current target value is indirectly depending on itself.

At each step, newly generated analysis variables are added to the worklist whenever encountered. The `VariableIndex`, as well as the `Dependencies`, are updated accordingly.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Solver.hs`

4.6 Modelling Arrays, Structs, and Unions

This section describes how arrays, structs, and unions are modelled in the current state of the framework. This is a bit tricky since in C/C++ (and thus in INSPIRE) all three structures can be combined with each other, multiple times. For example having an array of structs or a struct embedded in a struct embedded in a union, and so on.

The two main building blocks we need are `DataPath` and `FieldIndex`. A `FieldIndex` is used to reference a field in case of a struct or union, or an element (index) in case of an array. `DataPath` allows us to reference elements of multi-level structures using a list of `FieldIndex`. Both of these building blocks are implemented in a generic way to be easily extensible in the future. Thus `FieldIndex` is a type-class for which `SimpleFieldIndex` is our main implementation.

```haskell
class (Eq v, Ord v, Show v, Typeable v, NFData v) => FieldIndex v where
    -- ...

data SimpleFieldIndex = Field String
                      | Index Int
                      | UnknownIndex

data Symbol = Constant NodeAddress
            | Variable NodeAddress

instance FieldIndex SimpleFieldIndex where
-- ...

type SymbolicFormula = Ar.Formula CInt Symbol
```

The data structure `Symbol` is used together with `CInt` to specialise the generic arithmetic formula data structure (seeArithmetic Analysis). The result, `SymbolicFormula`, is used when acessing an element of an array.

```haskell
data DataPath i = Root
                | Narrow [i]
                | Expand [i]
                | Invalid
```

`DataPath` is used to access the underlying structure of *structured* memory locations, where `Root` refers to the whole structure and `Narrow` to a path starting at the root pointing to a specific sub-structure. `Expand` is the inverse to `Narrow` and is used to navigate from a sub-structure to an enclosing structure (Jordan 2014, 133–34).

We now combine these building blocks with the `ComposedValue` type-class.

```haskell
-- c...composed value type, for modelling the value of composed bojects
--      like arrays, structs, ...
-- i...field index type
-- v...leaf value type to be covered

class (Solver.ExtLattice c, FieldIndex i, Solver.ExtLattice v)
    => ComposedValue c i v | c -> i v where

toComposed :: v -> c
toValue    :: c -> v

setElement :: DataPath i -> c -> c -> c
getElement :: DataPath i -> c -> c

composeElements :: [(i,c)] -> c

top :: c
```

Note the functional dependency `c -> i v` stating that `i` and `v` are uniquely determined by `c`. Take a look at (Jones 2000) for more information on the topic of functional dependencies. An instance is provided for our use in the framework.

```haskell
data ValueTree i a = Leaf a
                   | Node (Data.Map.Map i (Tree i a))
                   | Empty
                   | Inconsistent

instance (FieldIndex i, Solver.ExtLattice a) => ComposedValue (Tree i a) i a where
-- ...
```

Each level in this `ValueTree` data structure corresponds to one level in the structured memory. Let us look at an example.

```c
int a[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

```haskell
let a = composeElements [
  (Index 0, composeElements [ (Index 0, toComposed 1),
                              (Index 1, toComposed 2),
                              (Index 2, toComposed 3) ]),
  (Index 1, composeElements [ (Index 0, toComposed 4),
                              (Index 1, toComposed 5),
                              (Index 2, toComposed 6) ])]
```

On the left we see the original structure from an input program, on the right the corresponding`ValueTree` instance. Since we are representing a 2D array, the `ValueTree` instance also has two levels and `Index` is used for field indexing (see definition of the `SimpleFieldIndex` type in [Datapath]). One more example using a struct follows.

```c
struct vector3 {
    int x;
    int y;
    int z;
};

struct vector3 u = {7, 8, 9};
```

```haskell
let u = composeElements [ (Field "x", toComposed 7),
                          (Field "y", toComposed 8),
                          (Field "z", toComposed 9) ]
```

Since we are using a struct instead of `Index` we use `Field` together with the corresponding field name as string.

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Entities/FieldIndex.hs

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Framework/PropertySpace/ComposedValue.hs

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Framework/PropertySpace/ValueTree.hs

4.7 Generic Data-Flow Analysis

The generic DFA is implemented by a combination of data structures, more specific analyses, and by the main function `dataflowValue`. Some of the data structures and more specific analyses are covered in Solver. The remaining analyses are covered afterwards and grouped under Base Analyses.

To easily manage the more specific analyses the `DataFlowAnalysis` data structure represents a specific instance of a DFA and is composed of the following fields.

```haskell
-- a...analysis type
-- v...property space

data DataFlowAnalysis a v = DataFlowAnalysis
    { analysis                    :: a,
    , analysisIdentifier          :: Solver.AnalysisIdentifier,
    , variableGenerator           :: NodeAddress -> Solver.TypedVar v,
    , topValue                    :: v,
    , entryPointParameterHandler :: NodeAddress -> Solver.TypedVar v,
    , initialValueHandler         :: NodeAddress -> v,
    , initValueHandler            :: v }
```

The `analysis` field is equivalent to the `aidToken` field in `AnalysisIdentifier`. It serves the purpose of distinguishing different analyses by their type. Yet, we also have to maintain the `AnalysisIdentifier` itself.

The `variableGenerator` is used for hooking the variable / constraint generator of the specific analysis into the generic framework. It follows the top value of the corresponding property space. As a safe fallback, the top value is used by the DFA whenever something unsupported is encountered.

The `entryPointParameterHandler` is used for covering parameters of program entry points – commonly `argc`, `argv`.

`initialValueHandler` and `initValueHandler` specify the respective value for memory locations – either upon an initialisation or program start. The `initialValueHandler` is consulted for dynamic memory locations, while `initValueHandler` deals with static memory locations.

While the data constructor `DataFlowAnalysis` is exported, a simplified version is also provided.

```haskell
mkDataFlowAnalysis :: (Typeable a, Solver.ExtLattice v)
                   => a
                   -> String
                   -> (NodeAddress -> Solver.TypedVar v)
                   -> DataFlowAnalysis a v
```

In this function the `analysisIdentifier` is populated with an `AnalysisIdentifier` constructed from the given analysis token and string. The third parameter is used as the variable / constraint generator. The respective top value can be deduced from the type and is outputted as result by all handlers, thus `entryPointParameterHandler`, `initialValueHandler`, and `initValueHandler` are set to yield the property space's top value.

An instance of `DataFlowAnalysis` can then be used with the main function of generic DFA `dataflowValue`. Its signature looks as follows.

```haskell
dataflowValue :: (ComposedValue.ComposedValue a i v, Typeable d)
              => NodeAddress
              -> DataFlowAnalysis d a
              -> [OperatorHandler a]
              -> Solver.TypedVar a
```

The first argument specifics the node for which a variable, representing the dataflow value, needs to be computed. Next an instance of `DataFlowAnalysis` is required to indicate the specific nature of the analysis. A list of `OperatorHandler`s (see Operator Handler) follows. The result is, of course, an analysis variable corresponding to the specific analysis. Note that `dataflowValue` can be specialised to a variable / constraint provider by fixing the second and third parameter.

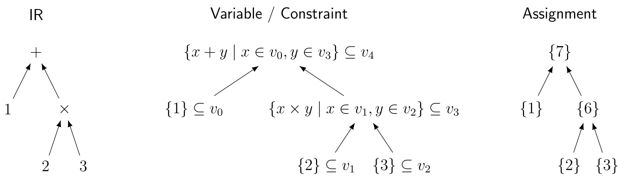


Figure 8: Illustrating dataflow value aggregation of `dataflowValue`.

Figure 8 illustrates how dataflow values are aggregated for the expression $1 + 2 \times 3$. As can be seen, the aggregation happens bottom-up.

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Framework/Dataflow.hs

4.7.1 Operator Handler

Before continuing on, the concept of `OperatorHandler` is presented. They provide a simple means to hook more specific logic into the generic DFA (or `ProgramPoint` analysis as we will see later).

This is used to interpret the semantics of abstract operators (and optionally derived operators) (see Extension Mechanism). The definition is straight forward and looks like this.

```haskell
data OperatorHandler a = OperatorHandler { covers    :: NodeAddress -> Bool,
                                         , dependsOn :: Assignment -> [Var],
                                         , getValue  :: Assignment -> a }
```

`covers` is used by the generic DFA to check whether this operator handler is to be used for the interpretation of a call to a given function symbol / operator. The other two functions house the logic of the handler itself.

4.8 Program Point

In the next two subsections we focus on the two interconnected concepts of program points and memory state. Both concepts have their dedicated generic variable / constraint generator which are utilised by the generic DFA.

The definition of program points follows.

```haskell
data Phase = Pre | Internal | Post

data ProgramPoint = ProgramPoint NodeAddress Phase
```

A `ProgramPoint` models a specific point in the execution of an INSPIRE program. For this purpose a `NodeAddress` referencing a specific node is used together with a `Phase`. The `Phase` describes at which phase of the execution the referenced node is referenced. It can be either

`Pre`
: before the referred node is processed;

`Internal`
: while the referred node is processed (only relevant for interpreting operators); or

`Post`
: after the referred node is processed.

4.8.1 Program Point Analysis

The purpose of the program point analysis is to derive properties from a given `ProgramPoint`. The generic variable / constraint generator function `programPointValue` of this analysis has the following signature.

```haskell
programPointValue :: (Solver.ExtLattice a)
                  => ProgramPoint
                  -> (ProgramPoint -> Solver.Identifier)
                  -> (ProgramPoint -> Solver.TypedVar a)
                  -> [OperatorHandler a]
                  -> Solver.TypedVar a
```

The result is a variable representing the requested information, for which the given `ProgramPoint` (first argument) represents a state value. The second argument is simply a variable id generator used to generate (unique) identifiers for analysis variables. The third argument is the analysis for which `programPointValue` is the base. The list of `OperatorHandler`s is used to intercept and interpret certain operators.

`programPointValue` utilises the predecessor analysis (covered in Predecessor Analysis). The following code example and figure illustrate how the INSPIRE tree is traversed by `programPointValue`.

```
// pseudo code
{
    a = 6;
    b = 7;
    f(a, b);
}
```

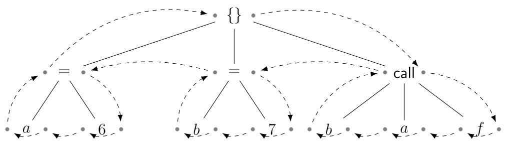


Figure 9: Illustrating INSPIRE backward traversal by `programPointValue`.

Each node in Figure 9 has two grey bullets, one to its left, the other to its right. They represent the `ProgramPoint`'s `Phase`s, `Pre` or `Post` respectively. These points are traversed in backward direction by the program point analysis – thus in reverse execution order. This concept is the basic building block for modelling the backwards analyses in the analysis framework.

Like the data-flow function, this is a generic base for analyses – not an analysis itself.

4.8.1.1 Data-Flow vs Program Point

The purpose of a data-flow analysis is to determine the value(s) a given expression may exhibit. In contrast to this, a program point analysis determines the state(s) of some (shared) structure at some point of program execution.

4.9 Memory State

A memory location is defined by a `NodeAddress` referencing its creation point. Furthermore, we can combine `MemoryLocation` with `ProgramPoint` to define ourselves `MemoryStatePoint`. A `MemoryStatePoint` references the state of a memory location at a given `ProgramPoint`.

```haskell
data MemoryLocation = MemoryLocation NodeAddress

data MemoryStatePoint = MemoryStatePoint ProgramPoint MemoryLocation
```

In order to find the state of a `MemoryLocation` at a given `ProgramPoint`, a reaching definitions analysis has to be conducted.

4.9.1 Reaching Definitions Analysis

A reaching definitions analysis is capable of identifying all definitions of a given memory location reaching the `ProgramPoint` of interest. We therefore now look at the definition of the reaching definitions analysis' property space.

```haskell
data Definition = Initial
                | Creation
                | Declaration NodeAddress
                | MaterializingCall NodeAddress
                | Assignment NodeAddress
                | Initialization NodeAddress

type Definitions = BSet.UnboundSet Definition

instance Lattice Definitions where
    bot = BSet.empty
    merge = BSet.union

instance ExtLattice Definitions where
    top = BSet.Universe
```

`Definition` models the different ways a memory location in INSPIRE can be initialised.

`Initial`
: The definition is obtained at program startup.

`Creation`
: The variable is left undefined at its point of allocation.

`Declaration`
: The definition is conducted by an assignment triggered through a *materializing*[11] declaration. The node address refers to that Declaration.

`MaterializingCall`
: The definition is conducted by an assignment triggered through a materializing call. The node address refers to that CallExpr.

`Assignment`
: The definition is conducted by an assignment. The node address refers to that assignment.

`Initialization`
: The definition is conducted by an initialisation expression. The node address refers to that initialisation.

[11] A temporary (r-value) is written to a memory location so that it can be referenced – constructor and destructor are managed accordingly – see the corresponding CLang documentation.

The signature of `reachingDefinitions` looks as follows:

```haskell
reachingDefinitions :: MemoryStatePoint -> Solver.TypedVar Definitions
```

This analysis is used by the memory state analysis.

4.9.2 Memory State Analysis

Finding the state of a `MemoryLocation` at a given `ProgramPoint` can now be achieved by the memory state analysis, which's variable / constraint generator builds on top of the reaching definitions analysis.

The analysis variable constructed by `reachingDefinitions`, for a given `MemoryState ms`, can be passed to the Solver's `get` function together with an `Assignment a` to yield the `Definitions`.

```haskell
reachingDefVar = reachingDefinitions ms
reachingDefVal = Solver.get a reachingDefVar
```

This is almost the information we want. From the we need to deduce the actual value the memory location exhibits. This is done by the following step.

```haskell
definingValueVars a = BSet.applyOrDefault [] (concat . (map go) . BSet.toList)
                    $ reachingDefVal a
  where
    go (Declaration       addr) = [variableGenerator analysis $ goDown 1 addr]
    go (MaterializingCall addr) = [variableGenerator analysis $ addr]
    go (Assignment        addr) = [definedValue addr ml analysis]
    go (Initialization    addr) = [definedValue addr ml analysis]
    go _                        = []
```

Here, `definedValue` models the underlying memory structure of the given `MemoryLocation ml` using an instance of the `ComposedValue` type-class. Note that neither for `Initial` nor for `Creation` we can deduce a value more accurate than the worst-case ⊤ element (line 8).

With this last piece in place we can define the memory state analysis' variable / constraint generator function `memoryStateValue`.

```haskell
1  memoryStateValue (MemoryStatePoint _ ml@(MemoryLocation loc)) analysis = var
2    where
3      var = Solver.mkVariable varId [con] Solver.bot
4      con = Solver.createConstraint dep val var
5  
6      dep a = (Solver.toVar reachingDefVar) : (
7                  if BSet.isUniverse defs || BSet.member Creation defs
8                  then []
9                  else (map Solver.toVar $ definingValueVars a)
10             )
11       where
12         defs = reachingDefVal a
13 
14     val a = case () of
15         _ | BSet.isUniverse defs      -> Solver.top
16           | BSet.member Initial defs  -> Solver.merge init value
17           | otherwise                 -> value
18       where
19         init = initialValueHandler analysis loc
20 
21         value = if BSet.member Creation $ defs
22                 then ComposedValue.top
23                 else Solver.join $ map (Solver.get a) (definingValueVars a)
24 
25         defs = reachingDefVal a
26 
27     varId = -- ..
```

The `initialValueHandler` is used in line 16 if the set of definitions `defs` contains an `Initial`. The `initialValueHandler`'s result is merged with deduced values. Constraints (and dependencies) are created accordingly.

4.10 Specific Analyses

Now, after covering the generic DFA, we look at more *specific analyses*. As already mentioned, these specialised analyses are used by the generic DFA. While the generic DFA covers the INSPIRE core language, the specific analyses cover individual language extensions. This relationship is illustrated in Figure 10.

As each language extension typically defines a new type and operators, each matching analysis defines an abstract value space for the values of expressions composed by the language extension. Furthermore, it models the semantics of the associated operators on top of these property spaces. Thereby we have the cases:

- Operators are abstract literals, hence `OperatorHandler` has to be defined.
- Operators are derived functions, for which their interpretation is implicitly covered by interpreting their definitions – or an `OperatorHandler` may be defined, if desired.

Another important aspect to note is that the property spaces of analyses are not unique. There are infinitely many different ones, differing in their correctness, accuracy and evaluation performance. The designer of an analyses has to chose the right one for a given task.

Also note that not all language extensions are clearly separated. In Insieme's current state, the *basic language extension* combines some frequently used language constructs like arithmetic and boolean.

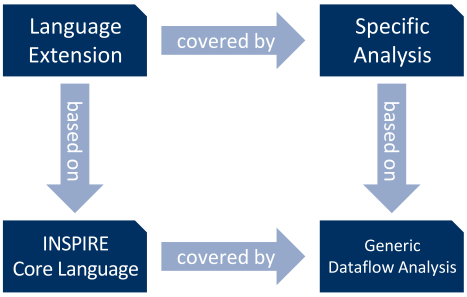


Figure 10: Connection between language a extension, INSPIRE core language, a specific analysis and the generic DFA.

4.10.1 Identifier Analysis

The purpose of the identifier analysis is to determine the value of expressions representing identifiers. Thus expressions of type `Identifier` (see [Language Basic]).

This analysis is used by the data path analysis (see Data Path Analysis) to track the field names of structs and unions. Its property space is defined as follows.

```haskell
    data Identifier = Identifier String
      deriving (Eq,Ord,Generic,NFData)

    instance Show Identifier where
        show (Identifier s) = s

    type IdentifierSet = BSet.UnboundSet Identifier

    instance Solver.Lattice IdentifierSet where
        bot   = BSet.empty
        merge = BSet.union

    instance Solver.ExtLattice IdentifierSet where
        top   = BSet.Universe
```

The bottom value of the property space represents no identifiers, while the top element represents all identifiers – hence `BSet.empty` and `BSet.Universe` are used respectively. When two data-flows merge, the resulting data-flow should contain all identifiers from the two branches, therefore `BSet.union` is used as merge operator. Also note that the identifier analysis is a *may analysis*.

For the analysis to be usable by the framework, a unique identifier is required. Together with this identifier we provide the constraint generator function `identifierValue`.

```haskell
1  data IdentifierAnalysis = IdentifierAnalysis
2    deriving (Typeable)
3
4  identifierValue :: NodeAddress -> Solver.TypedVar (ValueTree SimpleFieldIndex IdentifierSet)
5  identifierValue addr = case getNode addr of
6
7      IR.Node IR.Literal [_, IR.Node (IR.StringValue x) _] ->
8          Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton (Identifier x))
9
10     _ -> dataflowValue addr analysis []
11
12   where
13     analysis = mkDataFlowAnalysis IdentifierAnalysis "I" identifierValue
14     idGen    = mkVarIdentifier analysis
15     compose  = ComposedValue.toComposed
```

In line 13, a `DataFlowAnalysis` is instantiated using the unique identifier, a string `"I"` for easy identification of the associated analysis variables, and `identifierValue`, holding the main logic of the identifier analysis.

The interpretation of most IR constructs is handled by the generic DFA. We only need to cover the nodes relevant to us. In our case a Literal, where the second child is a StringValue[12]. An analysis variable is created, using `mkVariable` (line 8) and a newly generated identifier (line 14). No constraints are associated with the variable and the value associated with the variable is a (bound) set containing the identifier, wrapped up in a `ValueTree`.

One can also see that that no `OperatorHandler`s are used when invoking `dataflowValue` in line 10. This means that all information derived from Literal nodes is passed along the data-flow without any additional modifications.

4.10.1.1 Connection Between Generic DFA and Specialised Analysis

In line 10 we can see the interaction between the generic DFA and a specialised analysis. The user would simply invoke `identifierValue` and pass the result to the solver. If the node targeted is not a Literal, the input program falls back to the generic DFA, which calls our specialised analysis function `identifierValue` for the relevant nodes. The results are merged according to the `Lattice` instantiation.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Identifier.hs`

4.10.2 Arithmetic Analysis

Arithmetic Formulas has already presented the concept and implementation of arithmetic formulas. Their main use appears in the arithmetic analysis. Its task is to derive the arithmetic values a given expression may exhibit.

As mentioned in [Language Mechanism], arithmetic types and operators are defined in the *basic language extension*. The arithmetic analysis thus covers this part of the *basic language extension*.

The type alias `SymbolicFormula`, previously introduced in [Datapath], is used together with `BoundSet` to compose the arithmetic property space.

```haskell
type SymbolicFormulaSet b = BSet.BoundSet b SymbolicFormula

instance BSet.IsBound b => Solver.Lattice (SymbolicFormulaSet b)  where
    bot   = BSet.empty
    merge = BSet.union

instance BSet.IsBound b => Solver.ExtLattice (SymbolicFormulaSet b)  where
    top   = BSet.Universe
```

Similar to the identifiers analysis, the arithmetic analysis is a *may analysis*, therefore `BSet.empty`, `BSet.Universe`, and `BSet.union` are used for ⊥, ⊤, and the merge operator respectively.

An integer literal in the input program is translated to INSPIRE as a Literal containing a StringValue node. As the string contained needs to be parsed, a utility `parseInt` is provided. `parseInt` understands the different C/C++ integer literal formats (eg `0xf`, `072`, `1ul`, `-24`).

[12] Note that this is an simplification that does not cause harm since *all* literals are covered, where only those of the type 'identifier' would be required.

```haskell
data CInt = CInt32  Int32   -- ^ Represents @int@
          | CInt64  Int64   -- ^ Represents @long@
          | CUInt32 Word32  -- ^ Represents @unsigned int@
          | CUInt64 Word64  -- ^ Represents @unsigned long@
  deriving (Eq,Ord,Generic,NFData)

instance Show     CInt where -- ...
instance Num      CInt where -- ...
instance Enum     CInt where -- ...
instance Real     CInt where -- ...
instance Integral CInt where -- ...

parseInt :: String -> Maybe CInt
```

Before looking at the body of `arithmeticValue` we need to setup the variable / constraint generator implementing the arithmetic analysis, to be used by the solver.

```haskell
 1  data ArithmeticAnalysis = ArithmeticAnalysis
 2    deriving (Typeable)
 3
 4  analysis = (mkDataFlowAnalysis ArithmeticAnalysis "A" arithmeticValue) {
 5
 6          initialValueHandler = \a -> compose $ BSet.singleton
 7                                              $ Ar.mkVar
 8                                              $ Constant (Addr.getNode a) a,
 9
10          initValueHandler = compose $ BSet.singleton $ Ar.zero
11      }
12
13  idGen = mkVarIdentifier analysis
```

The default initial value handler yield the top element of the property space. For the arithmetic analysis this default is not desired, hence we override it in lines 6–10. For static memory locations (handled by `initValueHandler`) we can assume the memory location's value to be initialised to 0. For dynamic memory locations we handle the location as a variable, which enables us to reason about it in more detail than just assuming its content to be ⊤. For instance, we can deduce that the value of the memory location subtracted from itself yields 0 as result.

Now we are ready to look at `arithmeticValue`.

```haskell
 1  arithmeticValue :: Addr.NodeAddress
 2                  -> Solver.TypedVar (ValueTree SimpleFieldIndex
 3                                              (SymbolicFormulaSet BSet.Bound10))
 4
 5  arithmeticValue addr = case Addr.getNode addr of
 6
 7      IR.Node IR.Literal [t, IR.Node (IR.StringValue v) _] | isIntType t ->
 8        case parseInt v of
 9          Just cint -> Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton
10                                                                  $ Ar.mkConst cint)
11
12          Nothing   -> Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton
13                                                                  $ Ar.mkVar c
14            where
15              c = Constant (Addr.getNode addr) addr)
16
17      IR.Node IR.Variable _ | Addr.isLoopIterator addr ->
18          Solver.mkVariable (idGen addr) [] (compose $ BSet.Universe)
19
20      IR.Node IR.Variable (t:_) | isIntType t && isFreeVariable addr ->
21          Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton
22                                                     $ Ar.mkVar
23                                                     $ Variable (Addr.getNode addr) addr)
24
25      IR.Node IR.CastExpr (t:_) | isIntType t -> var
26        where
27          var = Solver.mkVariable (idGen addr) [con] Solver.bot
28          con = Solver.forward (arithmeticValue $ Addr.goDown 1 addr) var
29
30      _ -> dataflowValue addr analysis ops
31    where
32      -- ...
```

In case the given node address points to a Literal of type integer (line 7), the integer parser utility is consulted (line 8). If the parse was successful, the result is wrapped in a (constant) arithmetic formula, a (bound) set, and a `ValueTree` (lines 9 and 10). Otherwise the given node is handled as a mathematical constant and wrapped similarly (lines 12–15).

If we encounter a variable, which is a loop-iterator (line 17), we immediately return `Universe` as the range is commonly to big to consider all possible values in the analysis. Free variables of type integer are handled explicitly by `arithmeticValue` (line 20). Such an explicitly handled variable is wrapped in an arithmetic formula, a (bound) set, and a `ValueTree`. Other variables are covered by the default dataflow analysis.

A CastExpr of (target) type integer simply forwards the `arithmeticValue` of its argument. For all other nodes, the generic DFA is used, hooking the defined operator handlers and this analysis into the framework. The DFA also covers declarations, parameter and return value passing, closures, calls to unknown external functions, inter-procedural analyses, and memory locations.

Since all arithmetic operators are abstract, `OperatorHandler`s need to be defined, however their implementation is straightforward. The encountered operator is simply mapped to the corresponding arithmetic operation.

```haskell
ops = [add, mul, sub, div, mod]

add = OperatorHandler cov dep (val Ar.addFormula)
  where
    cov a = any (isBuiltin a) ["int_add", "uint_add", "gen_add"]
```

The result of each integer operation depends on the arithmetic value of its operands. Thus we invoke`arithmeticValue` on the left- and right-hand side (lines 1 and 2).

```haskell
1  lhs = arithmeticValue $ Addr.goDown 2 addr
2  rhs = arithmeticValue $ Addr.goDown 3 addr
3
4  val op a = compose $ (BSet.lift2 op) (extract $ Solver.get a lhs)
5                                       (extract $ Solver.get a rhs)
6
7  dep _ = Solver.toVar <$> [lhs, rhs]
8
9  compose = ComposedValue.toComposed
10 extract = ComposedValue.toValue
```

For division, we first need to check if it is safe to divide using the two operands (seeArithmetic Formulas). If possible, the division is applied, otherwise `Universe` is used as result.

```haskell
div = OperatorHandler cov dep val
  where
    cov a = any (isBuiltin a) [ "int_div", "uint_div", "gen_div" ]
    val a = compose $ tryDiv (extract $ Solver.get a lhs) (extract $ Solver.get a rhs)

    tryDiv x y = if not (BSet.isUniverse prod) && all (uncurry Ar.canDivide)
                                                      (BSet.toList prod)
                  then BSet.map (uncurry Ar.divFormula) prod
                  else BSet.Universe
      where
        prod = BSet.cartProduct x y
```

We have already described how an arithmetic formula is passed from Haskell back to C++ inArithmetic Formulas. We therefore now look at making the arithmetic analysis available to C++.

```haskell
1  foreign export ccall "hat_arithmetic_value"
2    arithValue :: StablePtr Ctx.Context
3               -> StablePtr Addr.NodeAddress
4               -> IO (Ptr CArithmeticSet)
5
6  arithValue ctx_hs expr_hs = do
7      ctx  <- deRefStablePtr ctx_hs
8      expr <- deRefStablePtr expr_hs
9      let (res,ns) = Solver.resolve (Ctx.getSolverState ctx) (Arith.arithmeticValue expr)
10     let results = ComposedValue.toValue res
11     let ctx_c = Ctx.getCContext ctx
12     ctx_nhs <- newStablePtr $ ctx { Ctx.getSolverState = ns }
13     updateContext ctx_c ctx_nhs
14     passFormulaSet ctx_c $ BSet.map (fmap SymbolicFormula.getAddr) results
```

`arithValue`, exported as `hat_arithmetic_value`, receives a Context and an expression. The arithmetic analysis is invoked in line 9. The initial state of the solver is taken from the received Context. A new solver state `ns`, as well as the result of the analysis`res` is returned by `Solver.resolve`. The Context is updated with the new solver state in line 13, while the result is passed to C++ using `passFormulaSet`.

As the arithmetic analysis yields a (bound) set of arithmetic formulas, we pass this whole set via the FFI. Since we already know how a single formula is passed, we can easily build upon this knowledge and define ourselves `passFormulaSet`.

```haskell
foreign import ccall "hat_mk_arithmetic_set"
  arithmeticSet :: Ptr (Ptr CArithmeticFormula) -> CInt -> IO (Ptr CArithmeticSet)

passFormulaSet :: Integral c
               => Ctx.CContext
               -> BSet.BoundSet bb (Ar.Formula c Addr.NodeAddress)
               -> IO (Ptr CArithmeticSet)

passFormulaSet _ BSet.Universe = arithmeticSet nullPtr (-1)
passFormulaSet ctx_c bs = do
    formulas <- mapM (passFormula ctx_c) (BSet.toList bs)
    withArrayLen' formulas arithmeticSet
  where
    withArrayLen' :: Storable a => [a] -> (Ptr a -> CInt -> IO b) -> IO b
    withArrayLen' xs f = withArrayLen xs (\s a -> f a (fromIntegral s))
```

The imported function `hat_mk_arithmetic_set` simply takes an array of arithmetic formulas (C++) and combines them into an`ArithmeticSet`, the C++ equivalent to `SymbolicFormulaSet`. A negative length (second argument) indicates that the set is `Universe`.

`passFormulaSet` passes every formula of the arithmetic set from Haskell to C++ first, and uses`arithmeticSet` to combine them to an `ArithmeticSet` (C++).

The arithmetic analysis is provided in C++:

```C++
extern "C" {
    ArithmeticSet* hat_arithmetic_value(StablePtr ctx, const HaskellNodeAddress expr_hs);
}

ArithmeticSet getArithmeticValue(Context& ctxt, const ExpressionAddress& expr) {
    auto expr_hs = ctxt.resolveNodeAddress(expr);
    ArithmeticSet* res_ptr = hat_arithmetic_value(ctxt.getHaskellContext(), expr_hs);
    ArithmeticSet res(std::move(*res_ptr));
    delete res_ptr;
    return res;
}
```

code/analysis/src/cba/haskell/arithmetic_analysis.cpp

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Adapter.hs

code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Arithmetic.hs

4.10.3 Boolean Analysis

Next, the boolean analysis is presented. Its task is to describe the truth value an expression might have during execution. For the property space we define the possible outcomes, together with the `Lattice` instance.

```haskell
data Result = AlwaysTrue
            | AlwaysFalse
            | Both
            | Neither
  deriving (Eq,Enum,Show,Generic,NFData)

instance Solver.Lattice Result where
    bot = Neither

    merge Neither x = x
    merge x Neither = x
    merge x y | x == y = x
    merge _ _ = Both

instance Solver.ExtLattice Result where
    top = Both
```

If the analysis can determine that the boolean value of an expression is constant (ie is not input dependent), the result is either`AlwaysTrue` or `AlwaysFalse`. If the boolean value is not constant, the result is`Both`. `Neither` is the initial state, until more accurate information regarding the analysed expressions could be determined. Also, it is the value of expressions which are never evaluated (dead code) due to the implicit dead-code detection of our framework.

Again, we have to setup an identifier for the analysis and provide the main logic of the analysis.

```haskell
data BooleanAnalysis = BooleanAnalysis
  deriving (Typeable)

booleanValue :: NodeAddress -> Solver.TypedVar (ValueTree SimpleFieldIndex Result)
booleanValue addr =
    case () of _
                | isBuiltin addr "true"  -> Solver.mkVariable (idGen addr) []
                                            $ compose AlwaysTrue
                | isBuiltin addr "false" -> Solver.mkVariable (idGen addr) []
                                            $ compose AlwaysFalse
                | otherwise              -> dataflowValue addr analysis ops
  where
    compose = ComposedValue.toComposed
    extract = ComposedValue.toValue

    analysis = mkDataFlowAnalysis BooleanAnalysis "B" booleanValue
    idGen = mkVarIdentifier analysis

    ops = [ lt, le, eq, ne, ge, gt ]

    le = OperatorHandler cov dep (val cmp)
      where
        cov a = any (isBuiltin a) ["int_le", "uint_le"]
        cmp x y = case numCompare x y of
            NumEQ     -> AlwaysTrue
            NumLT     -> AlwaysTrue
            Sometimes -> Both
            _         -> AlwaysFalse

    -- lt, eq, ne, ge and gt following the same pattern as le

    lhs = arithmeticValue $ goDown 2 addr
    rhs = arithmeticValue $ goDown 3 addr

    dep a = Solver.toVar <$> [lhs, rhs]

    val op a = combine (extract $ Solver.get a lhs) (extract $ Solver.get a rhs)
      where
        combine BSet.Universe _ = compose Both
        combine _ BSet.Universe = compose Both
        combine x y = compose . Solver.join $ [ u `op` v | u <- BSet.toList x
                                                         , v <- BSet.toList y ]
```

Initially, we check if a given node is equivalent to the operators`true` or `false`. If so, we wrap `AlwaysTrue` or `AlwaysFalse` in a `ValueTree` and return an analysis variable with it as initial value. Otherwise we resort to the generic DFA.

The `OperatorHandler`s `lt`, `le`, `eq`, `ne`, `ge`, and `gt` are setup to cover their respective comparators in lines 21–30. Furthermore, we have to state the dependency (lines 32–35) between the left and right hand side's arithmetic value and the current expression.

`arithmeticValue`, part of the arithmetic analysis (see Arithmetic Analysis), yields a (bound) set of arithmetic formulas representing the arithmetic values of an expression. The `OperatorHandler` defined in line 21 covers the operators `int_le` and `uint_le`, which are used in INSPIRE for the (unsigned) integer comparison *less than or equal*. The remaining `OperatorHandler`s are defined similarly. Together with `val`, defined in line 37, it

- extracts the (bound) sets of arithmetic formulas for the left and right hand sides;
- compares each element of the `lhs`'s set with each element of the `rhs`'s set, using `numCompare` (see Arithmetic Formulas); and
- maps the result of each comparison to the corresponding analysis outcome, defined by the`OperatorHandler`;

It is important not that the logical operations `bool_not`, `bool_and`, and `bool_or` are defined in the *basic language extension* using IfStmts and closures to correctly model short-circuit-evaluation of C/C++. We do not need to handle this explicitly thanks to the framework.

This analysis is made available to Insieme via FFI.

```haskell
foreign export ccall "hat_check_boolean"
  checkBoolean :: StablePtr Ctx.Context -> StablePtr Addr.NodeAddress -> IO CInt

checkBoolean ctx_hs expr_hs = do
    ctx  <- deRefStablePtr ctx_hs
    expr <- deRefStablePtr expr_hs
    let (res,ns) = Solver.resolve (Ctx.getSolverState ctx) $ booleanValue expr
    let ctx_c = Ctx.getCContext ctx
    ctx_nhs <- newStablePtr $ ctx { Ctx.getSolverState = ns }
    updateContext ctx_c ctx_nhs
    evaluate $ fromIntegral $ fromEnum $ ComposedValue.toValue res
```

The structure of `checkBoolean` is quite similar to `arithValue`, defined Arithmetic Analysis. It is sufficient to use a `CInt` for the boolean analysis' result as it is just an enumeration. The conversion happens by combining `fromIntegral`, `fromEnum`, and `ComposedValue.toValue`.

In C++, `hat_check_boolean` is imported and wrapped.

```C++
 1  extern "C" {
 2      int hat_check_boolean(StablePtr ctxt, const HaskellNodeAddress expr_hs);
 3  }
 4
 5  enum class BooleanAnalysisResult : int {
 6      AlwaysTrue,
 7      AlwaysFalse,
 8      Both,
 9      Neither
10  };
11
12  BooleanAnalysisResult checkBoolean(Context& ctxt, const ExpressionAddress& expr) {
13      auto expr_hs = ctxt.resolveNodeAddress(expr);
14      int res_hs = hat_check_boolean(ctxt.getHaskellContext(), expr_hs)
15      auto res = static_cast<BooleanAnalysisResult>(res_hs);
16      if(res == BooleanAnalysisResult::Neither) {
17          std::vector<std::string> msgs{"Boolean Analysis Error"};
18          throw AnalysisFailure(msgs);
19      }
20      return res;
21  }
```

The result of `hat_check_boolean` is received as integer and mapped to the corresponding enum value in line 15. The following four convenience functions are provided.

```C++
bool isTrue(Context& ctxt, const ExpressionAddress& expr) {
    return checkBoolean(ctxt, expr) == BooleanAnalysisResult::AlwaysTrue;
}

bool isFalse(Context& ctxt, const ExpressionAddress& expr) {
    return checkBoolean(ctxt, expr) == BooleanAnalysisResult::AlwaysFalse;
}

bool mayBeTrue(Context& ctxt, const ExpressionAddress& expr) {
    auto res = checkBoolean(ctxt, expr);
    return res == BooleanAnalysisResult::AlwaysTrue
        || res == BooleanAnalysisResult::Both;
}

bool mayBeFalse(Context& ctxt, const ExpressionAddress& expr) {
    auto res = checkBoolean(ctxt, expr);
    return res == BooleanAnalysisResult::AlwaysFalse
        || res == BooleanAnalysisResult::Both;
}
```

Note that for clarity `BooleanAnalysisResult` has been defined in C++ *and* Haskell. In reality, a C enum is defined in a header file which is imported in C++ and Haskell (via `c2hs`). This way they share the exact same definition and are thus implicitly consistent.

🗎 `code/analysis/src/cba/haskell/boolean_analysis.cpp`

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Adapter.hs`

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Boolean.chs`

4.10.4 Data Path Analysis

`DataPath`s have been covered in [Datapath]. The framework also contains an analysis for deducing the (bound) set of possible `DataPath` values for a given expression. This analysis is used by the reference analysis (see Reference Analysis).

As we can use the `DataPath` entity directly we can focus immediately on the lattice instance.

```haskell
type DataPathSet i = BSet.UnboundSet (DataPath i)

instance (FieldIndex i) => Solver.Lattice (DataPathSet i) where
    bot   = BSet.empty
    merge = BSet.union

instance (FieldIndex i) => Solver.ExtLattice (DataPathSet i) where
    top = BSet.Universe
```

Note that the data path analysis is, again, a *may analysis*.

`dataPathValue` does not need to handle specific types of nodes, it is only concerned with operators as it covers the datapath language extension[13].

[13] 🗎 `code/core/include/insieme/core/lang/datapath.h`

We therefore only have to define the relevant `OperatorHandler`s. The following operators are handled:

`dp_root`

This operator indicates the root of the object, hence `DataPath.root` is wrapped and returned.

`dp_member`

The identifier analysis is used to deduce the target field name, which is then used to modify the `DataPath`. One `DataPath.step` is added using the field name.

`dp_element`

Causes a recursion to find the *nested path* within and uses the arithmetic analysis to find the index used when calling the operator. Both results are then merged.

`dp_component`

Same as `dp_element`.

Note that this analysis is currently not accessible from C++.

🗎 `code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/DataPath.hs`

4.10.5 Reference Analysis

Upon encountering a reference during an analysis, we are often interest in the location this reference points to. The reference analysis is tasked with answering this question. Note that the reference language extension is covered by [Language Mechanism].

In the common case, a reference can be defined by the creation point of the (root) object and a `DataPath` to reference a sub-object within. The other two edge cases include `NULL` references and uninitialised references. This composes our property space.

```haskell
data Reference i = Reference { creationPoint :: NodeAddress
                             , dataPath      :: DP.DataPath i }
                 | NullReference
                 | UninitializedReference
  deriving (Eq,Ord,Show,Generic,NFData)

type ReferenceSet i = BSet.UnboundSet (Reference i)

instance (FieldIndex i) => Lattice (ReferenceSet i) where
    bot   = BSet.empty
    merge = BSet.union

instance (FieldIndex i) => ExtLattice (ReferenceSet i) where
    top = BSet.singleton UninitializedReference
```

Again we have a .

Next we take a look at the corresponding constraint generator function.

```haskell
data ReferenceAnalysis = ReferenceAnalysis
  deriving (Typeable)

referenceValue :: (FieldIndex i)
               => NodeAddress
               -> TypedVar (ValueTree i (ReferenceSet i))

referenceValue addr = case getNodeType addr of
    IR.Literal ->
        mkVariable (idGen addr) [] $ compose
                                   $ BSet.singleton
                                   $ Reference (crop addr) DP.root

    IR.Declaration | isMaterializingDeclaration (Addr.getNode addr) ->
        mkVariable (idGen addr) [] $ compose
                                   $ BSet.singleton
                                   $ Reference addr DP.root

    IR.CallExpr | isMaterializingCall (Addr.getNode addr) ->
        mkVariable (idGen addr) [] $ compose
                                   $ BSet.singleton
                                   $ Reference addr DP.root

    _ -> dataflowValue addr analysis opsHandler
  where
    analysis = (mkDataFlowAnalysis ReferenceAnalysis "R" referenceValue){
        entryPointParameterHandler = epParamHandler,
        initValueHandler = compose $ BSet.singleton $ NullReference }
    idGen = mkVarIdentifier analysis
    compose = ComposedValue.toComposed
    -- ...
```

As can be seen, the main node types we are interested in are Literals, Declarations and CallExprs where the latter two are only relevant when they are materializing. This is due to the fact that these are the only occurrences where memory is allocated.

The operators defined by the reference language extension are covered by the`OperatorHandler`s. The following list describes how each of them are handled. Note that we do not have to handle derived operators as their interpretation is covered by their definition.

`ref_alloc`
: The reference points to a newly allocated object, we can therefore use the current node address as creation point and `DataPath.root` to model the resulting reference.

`ref_decl`
: The target reference can be modelled by using the enclosing Declaration of this node as creation point. Again we use to refer to the whole object.

`ref_null`
: Yields a `NullReference`.

`ref_narrow`
: Since this operator is used to obtain a reference to a sub-object within a referenced object, we not only have to initiate another reference analysis for the referenced object, but also combine the `DataPath`(s) of that analysis' results with the `DataPath` provided to the invocation of `ref_narrow`.

`ref_expand`
: Handled similarly to `ref_narrow`, just expanding instead of narrowing the `DataPath` of references.

`ref_cast`
: Causes a recursive call of the reference analysis to the referenced object.

`ref_reinterpret`
: Same as `ref_cast`.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Reference.hs`

`code/core/include/insieme/core/lang/reference.h`

4.10.6 Callable Analysis

The callable analysis is a special case of specific analyses as it does not cover a language extension. Instead it covers function types, which are part of the INSPIRE core language, together with the related node types Literal, Lambda, and BindExpr.

It is used to identify the target function of a call expression. The three different callable types are Lambda, Literal, and Closure. We therefore have the following property space:

```haskell
data Callable = Lambda NodeAddress
              | Literal NodeAddress
              | Closure NodeAddress
  deriving (Eq,Ord,Generic,NFData)

type CallableSet = BSet.UnboundSet Callable

instance Solver.Lattice CallableSet where
    bot   = BSet.empty
    merge = BSet.union

instance Solver.ExtLattice CallableSet where
    top   = BSet.Universe
```

The analysis is straight forward in that it handles the related nodes.

```haskell
data CallableAnalysis = CallableAnalysis
  deriving (Typeable)

callableValue :: NodeAddress -> Solver.TypedVar (ValueTree SimpleFieldIndex CallableSet)
callableValue addr = case getNodeType addr of
    IR.LambdaExpr ->
        Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton
                                                    $ Lambda (fromJust $ getLambda addr))

    IR.LambdaReference ->
        Solver.mkVariable (idGen addr) [] (compose $ getCallables4Ref addr)

    IR.BindExpr ->
        Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton (Closure addr))

    IR.Literal ->
        Solver.mkVariable (idGen addr) [] (compose $ BSet.singleton (Literal addr))

    _ -> dataflowValue addr analysis []
  where
    analysis = mkDataFlowAnalysis CallableAnalysis "C" callableValue
    idGen = mkVarIdentifier analysis
    compose = ComposedValue.toComposed
    getCallables4Ref = -- ...
```

The only special case is encountered when dealing with a LambdaReference. In this case the tree is searched upwards from the LambdaReference for a matching LambdaDefinition. If the search was successful the Lambda contained in the LambdaDefinition is used, otherwise the property space's ⊤ value is returned. This is achieved by `getCallables4Ref`. This is indicating that any unknown target function may be represented by the targeted expression.

🗎 code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Callable.hs

4.11 Helper Analyses

The analyses described in this section are different from the previously described specific analyses in that they do not cover a certain language extension (or INSPIRE core language construct) and are not based on the generic DFA. However, they are supporting the definition of the generic DFA as well as the specific analyses.

4.11.1 Reachability Analysis

The reachability analysis is divided into two parts, `reachableIn` and `reachableOut`. Despite both of them being distinct analyses with their own respective analysis variables, they work together in tandem to answer the question of reachability. In detail, `reachableIn` indicates whether the beginning of the given node is reachable. In contrast to this, `reachableOut` indicates whether the end of a given node is reachable during some execution trace.

For instance, the beginning of a statement inside a CompoundStmt is reachable iff the end of the previous statement is reachable. If there is no previous statement within the CompoundStmt, the reachability depends on whether the beginning of the CompoundStmt is reachable.

The property space definitions for both analyses are trivial:

```haskell
newtype Reachable = Reachable Bool
  deriving (Eq,Show,Generic,NFData)

instance Solver.Lattice Reachable where
    bot = Reachable False
    merge (Reachable a) (Reachable b) = Reachable $ a || b
```

Note that this definition for the property space is sufficient and does not require to be an `ExtLattice` as this analysis is not a DFA.

It follows the definition of the analysis identifier token:

```haskell
data ReachableInAnalysis = ReachableInAnalysis
    deriving (Typeable)

reachableInAnalysis :: Solver.AnalysisIdentifier
reachableInAnalysis = Solver.mkAnalysisIdentifier ReachableInAnalysis "R[in]"

data ReachableOutAnalysis = ReachableOutAnalysis
  deriving (Typeable)

reachableOutAnalysis :: Solver.AnalysisIdentifier
reachableOutAnalysis = Solver.mkAnalysisIdentifier ReachableOutAnalysis "R[out]"
```

4.11.1.1 Reachable In

We now look at the definitions of the constraint generators `reachableIn` and `reachableOut`. As both of them are too long to be displayed easily in this thesis as a whole, they are broken up into their logical parts.

```haskell
1  reachableIn :: NodeAddress -> Solver.TypedVar Reachable
2  reachableIn a | isRoot a = Solver.mkVariable (reachableInIdGen a) [] (Reachable True)
3  reachableIn a = case getNodeType parent of
4      -- ...
5    where
6      parent = fromJust $ getParent a
7      idGen = Solver.mkIdentifierFromExpression reachableInAnalysis a
8      compose = ComposedValue.toComposed
```

We define the beginning of the root node of INSPIRE to be always reachable reflecting the premise that the analysed code fragment is indeed invoked. For a given node, not being the root, we need to distinguish a couple of cases, depending on the type of the parent node. All of the following code snippets (until reaching the definition of `reachableOut`) are replacing line 4.

```haskell
    IR.Lambda -> Solver.mkVariable (idGen a) [] (Reachable True)
```

Considering Lambdas to always be reachable is an over-approximation needed to reduce runtime complexity. Otherwise the necessity for a context sensitive reachability analysis would arise.

```haskell
    IR.CompoundStmt -> var
      where
        n = getIndex a
        var = Solver.mkVariable (idGen a) [con] Solver.bot
        con = if n == 0
                then Solver.forward (reachableIn parent) var
                else Solver.forward (reachableOut $ goDown (n-1) parent) var
```

As already mentioned by the introduction to this subsection, reachability inside a CompoundStmt depends on the previous node inside the CompoundStmt, or if there is no previous node, the reachability of the CompoundStmt itself.

```haskell
 1      IR.IfStmt -> var
 2        where
 3          var = Solver.mkVariable (idGen a) [con] Solver.bot
 4          con = case getIndex a of
 5              0 -> Solver.forward (reachableIn parent) var
 6
 7              1 -> Solver.forwardIf (compose Boolean.AlwaysTrue)
 8                                    (Boolean.booleanValue $ goDown 0 parent)
 9                                    (reachableOut $ goDown 0 parent)
10                                    var
11
12              2 -> Solver.forwardIf (compose Boolean.AlwaysFalse)
13                                    (Boolean.booleanValue $ goDown 0 parent)
14                                    (reachableOut $ goDown 0 parent)
15                                    var
16
17              _ -> error "index out of bound"
```

If the target node is the *then* or *else* branch of an IfStmt, the reachability depends on the boolean value of the condition. Therefore we need to conduct a boolean analysis as can be seen in lines 8 and 13. The condition is considered reachable if the IfStmt itself is reachable.

```haskell
    IR.WhileStmt -> var
      where
        var = Solver.mkVariable (idGen a) [con] Solver.bot
        con = case getIndex a of
            0 -> Solver.forward (reachableIn parent) var

            1 -> Solver.forwardIf (compose Boolean.AlwaysTrue)
                                  (Boolean.booleanValue $ goDown 0 parent)
                                  (reachableOut $ goDown 0 parent)
                                  var

            _ -> error "index out of bound"
```

Regarding reachability, handling a WhileStmt is identical to handling an IfStmt except that there is no else branch to worry about.

```haskell
    _ -> var
      where
        var = Solver.mkVariable (idGen a) [con] Solver.bot
        con = Solver.forward (reachableIn parent) var
```

For all other nodes, the reachability of the target is equal to the reachability of the parent node.

4.11.1.2 Reachable Out

We now continue with the definition of `reachableOut`.

```haskell
1  reachableOut :: NodeAddress -> Solver.TypedVar Reachable
2  reachableOut a = case getNodeType a of
3      -- ...
4    where
5      idGen a = Solver.mkIdentifierFromExpression reachableOutAnalysis a
```

For the main part we require a case distinction. The following code snippets replace line 3.

```haskell
    IR.ReturnStmt -> Solver.mkVariable (idGen a) [] Solver.bot

    IR.ContinueStmt -> Solver.mkVariable (idGen a) [] Solver.bot

    IR.BreakStmt -> Solver.mkVariable (idGen a) [] Solver.bot
```

The three simple cases are ReturnStmt, ContinueStmt, and BreakStmt. As they directly break the control flow, we can say that the end of these statements is never reachable, hence the use of `Solver.bot`, corresponding to `Reachable False`.

```haskell
    IR.CompoundStmt -> var
      where
        var = Solver.mkVariable (idGen a) [cnt] Solver.bot
        cnt = if numChildren a == 0
                then Solver.forward (reachableIn a) var
                else Solver.forward (reachableOut $ goDown (numChildren a - 1) a) var
```

The end of a CompoundStmt is reachable iff the end of the last statement in it is reachable. Note that (for simplicity) we ignore the possibility of exceptions being raised on destructors, leading to an over-approximation in the result.

```haskell
    IR.IfStmt -> var
      where
        var = Solver.mkVariable (idGen a) [t,e] Solver.bot
        t = Solver.forward (reachableOut $ goDown 1 a) var
        e = Solver.forward (reachableOut $ goDown 2 a) var
```

In case of an IfStmt, the end is reachable, iff the end of one of the branches is reachable.

```haskell
   IR.WhileStmt -> var
      where
        var = Solver.mkVariable (idGen a) [cnt] Solver.bot
        cnt = Solver.forwardIf (ComposedValue.toComposed Boolean.AlwaysFalse)
                               (Boolean.booleanValue $ goDown 0 a)
                               (reachableOut $ goDown 0 a)
                               var
```

The end of a WhileStmt is reachable iff the condition can evaluate to false. Again, we therefore require the boolean analysis.

```haskell
    _ -> var
      where
        var = Solver.mkVariable (idGen a) [con] Solver.bot
        con = Solver.forward (reachableIn a) var
```

All other nodes follow the same schema where the end is reachable iff the beginning is reachable.

🗎 code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Reachable.hs

4.11.2 Exit Point Analysis

The exit point analysis tries to determine the possible exit points of a functions. These are typically ReturnStmts and the end of the function body. Reachability analysis (see [Reachability]) is used to determine whether the control flow can reach a potential exit point, or not.

The definition of the corresponding property space is trivial:

```haskell
newtype ExitPoint = ExitPoint NodeAddress
 deriving (Eq,Ord,Generic,NFData)

type ExitPointSet = BSet.UnboundSet ExitPoint

instance Solver.Lattice ExitPointSet where
    bot   = BSet.empty
    merge = BSet.union
```

Similar to the reachability analysis (previous section), the exit point analysis is not a DFA and therefore does not require the instantiation of an `ExtLattice`.

Only BindExpr and Lambda are of interest to this analysis. All other node types yield an empty set of`ExitPoint`s. For a BindExpr the exit point is simply the nested CallExpr. For a given Lambda, all ReturnStmts are collected and checked for reachability. All reachable exit points are joined in an `ExitPointSet`.

🗎 code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/ExitPoint.hs

4.11.3 Predecessor Analysis

The traversal of `ProgramPoint`s has already been illustrated in [Programpoint Traversal]. The predecessor analysis models this traversal accordingly. It is split into three main parts: the first part handles traversal from a `Pre ProgramPoint` to its previous `Post ProgramPoint`, the second deals with `Internal ProgramPoint`s, and the last one is concerned with the predecessor of `Post ProgramPoint`s.

For a given `ProgramPoint` the list of predecessors is computed and represented as constraints in our framework. The property space simply consists of a list of `ProgramPoint`s and concatenation.

```haskell
type PredecessorList = [ProgramPoint]

instance Solver.Lattice PredecessorList where
    join [] = []
    join xs = foldr1 (++) xs
```

For the preprocessor analysis we do not use a (bound) set, since for each node, there only exist a limited number of possible predecessors.

The definition of the corresponding analysis identifier and the signature of the analysis' constraint generator function follows.

```haskell
data PredecessorAnalysis = PredecessorAnalysis
  deriving (Typeable)

predecessorAnalysis :: Solver.AnalysisIdentifier
predecessorAnalysis = Solver.mkAnalysisIdentifier PredecessorAnalysis "pred_of"

predecessor :: ProgramPoint -> Solver.TypedVar PredecessorList
```

The three parts mentioned can be distinguished by pattern matching on the `ProgramPoint` parameter of the function. The base case for this traversal is encountered when asking for the predecessors of the root's `Pre ProgramPoint`. Since there is no execution step before this program point, the resulting analysis variable is bound to the empty list.

```haskell
predecessor p@(ProgramPoint a Pre) | isRoot a = Solver.mkVariable (idGen p) [] []
predecessor p@(ProgramPoint a Pre) = case getNodeType parent of
    -- handle part 1

predecessor p@(ProgramPoint a Internal) = case getNodeType a of
    -- handle part 2

predecessor p@(ProgramPoint a Post) = case getNodeType a of
    -- handle part 3


idGen :: ProgramPoint -> Solver.Identifier
idGen pp = Solver.mkIdentifierFromProgramPoint predecessorAnalysis pp
```

The second part, concerned with `Internal ProgramPoint`s, utilises the exit point Analysis (see Exit Point Analysis) and callable analysis (see Callable Analysis) to link to exit points of potential target functions.

An `Internal ProgramPoint` is returned by the third part only for CallExprs. All other node types either yield a `Pre ProgramPoint` if they are a basic expression, like Variable or Literal, or the corresponding `Post ProgramPoint` otherwise.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/Predecessor.hs`

4.11.4 Call Site Analysis

The call site analysis identifies all possible call sites for a given Lambda or BindExpr. We can use node addresses to reference a call site, the lattice definition follows:

```haskell
newtype CallSite = CallSite NodeAddress
  deriving (Eq,Ord,Generic,NFData)

instance Show CallSite where
    show (CallSite na) = "Call@" ++ (prettyShow na)

type CallSiteSet = BSet.UnboundSet CallSite

instance Solver.Lattice CallSiteSet where
    bot   = BSet.empty
    merge = BSet.union
```

Call sites can only be dervied for Lambda and BindExpr nodes, and only when it is not the root node. In the simple case all call sites are statically determined. Otherwise, we have to utilise the callable analysis (see Callable Analysis) to identify all potential call sites.

`code/analysis/src/cba/haskell/insieme-hat/src/Insieme/Analysis/CallSite.hs`

5 Constructing a new Analysis

In this chapter we construct a new analysis using the established framework. Our goal is to implement an *array out-of-bounds* analysis. We go through the construction, implementation, and integration into INSIEME step by step:

- Defining the goals of the new analysis.
- Investigating the relevant constructs and language extensions of the INsieme Parallel Intermediate REpresentation (INSPIRE).
- Deriving from the gathered information what components / additional analyses are required to fulfil our goal.
- Defining the property spaces of the analyses to implement.
- Defining the interface for communication between INSIEME and the Haskell-based Analysis Toolkit (HAT).
- Setting up the required boilerplate code.
- Writing a handful of test cases to allow for checking the implementation during development.
- Implementing the required analyses.
- Summarising what files have been added / modified by the process.

The implemented analysis is used to evaluate the current state of the framework.

5.1 Defining the Goals

The goal is to create an analysis which can identify out-of-bounds accesses to arrays. For simplicity we will only concern ourselves with accessing elements of an array beyond its upper bound. Also covering the lower bound will be a trivial extension and is left as an exercise to the reader.

The following code fragment will illustrate the essentials:

```C++
{ // Case 1
    int a[21];
    a[42];
}

{ // case 2
    int a[21];
    a[20];
}

{ // Case 3
    int a[21];
    int *ap = a + 10;
    ap[20];
}

{ // Case 4
    auto a = new int[21];
    a[42];
    delete[] a;
}

{ // Case 5
    int a[21];
    int b;
    a[b];
}

{ // Case 6
    int a[21];
    int b = a[42];
}
```

Case 1
: Accessing an element beyond the array's upper bound. The analysis returns `IsOutOfBounds` for `a[42]`.

Case 2
: Accessing an element within the array's upper bound. The analysis returns `IsNotOutOfBounds` for `a[20]`.

Case 3
: Same as Case 1, yet the array is accessed indirectly via a pointer offset to the base of the array. Same result as in Case 1.

Case 4
: Same as Case 1, yet the array is allocated using the `new` keyword. Same result as in Case 1.

Case 5
: Accessing the array with an undefined index, the analysis returns `MaybeOutOfBounds`.

Case 6
: Same as Case 1, but involves an assignment.

This marks the base for the following sections. Note that the framework will cover more complicated cases (eg where a pointer to an array is passed to a function and the function accesses the underlying array) automatically thanks to the generic Data-Flow Analysis (DFA).

5.2 Investigating INSPIRE

The different cases defined in the previous section are used as input code the Insieme compiler. The analysis runs on the resulting INSPIRE program, constructed by the frontend of the compiler. Using `insiemecc` we can feed each of the cases to the compiler and retrieve the corresponding Intermediate Representation (IR) in various formats. `case1.cpp` is the input to the compiler, while `case1.ir`, `case1.tree`, and `case1.json` compose the output we are interested in. The following command is used to obtain the files.

```
$ insiemecc --dump-tree case1.tree --dump-json case1.json --dump-ir case1.ir case1.cpp
```

`case1.ir` contains the pretty-printed text representation of the corresponding INSPIRE program. `case1.tree` can be viewed with a regular text editor to investigate the node structure of this INSPIRE program. Furthermore `case1.json` also contains the node structure, but in a format that can be viewed using INSPYER (see [INSPYER]), which provides a more interactive viewing experience.

Let us now take a look at the first case.

```
// case1.cpp

int main(void) {
    int a[21];
    a[42];
}
```

```
// case1.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main (){
    var ref<array<int<4>,21>,f,f,plain> v1 =
      ref_decl(type_lit(ref<array<int<4>,21>,f,f,plain>));
    *ptr_subscript(ptr_from_array(v1), 42);
    return 0;
}
```

```
// case1.tree

(Program | ...
    (CompoundStmt |
        (DeclarationStmt |
            (Declaration |
                (GenericType | ... )
                (CallExpr |
                    (GenericType |
                        (StringValue "ref")
                        (Parents )
                        (Types |
                            (GenericType |
                                (StringValue "array")
                                (Parents )
                                (Types |
                                    (GenericType | ... )
                                    (NumericType |
                                        (Literal |
                                            (GenericType | ... )
                                            (StringValue "21"))))
                                ... )))
                    (Literal |
                        (FunctionType | ... )
                        (StringValue "ref_decl"))
                    (Declaration | ... )))
            (Variable | ... ))
        (CallExpr |
            (GenericType |  ...)
            (Literal |
                (FunctionType | ... )
                (StringValue "ref_deref"))
            (Declaration |
                (GenericType | ... )
                (CallExpr |
                    (GenericType |  ...)
                    (LambdaExpr |
                        (FunctionType | ... )
                        (LambdaReference |
                            (FunctionType | ... )
                            (StringValue "ptr_subscript"))
                        (LambdaDefinition | ... ))
                    (Declaration | ... )
                    (Declaration | ... ))))
        (ReturnStmt | ... ))...)
```

As can be seen in `case1.ir` and `case1.tree`, the array is constructed using the operator `ref_decl` and accessed via the `ptr_subscript` operator. Note that the pointer subscript operation is *not* prohibited, however, dereferencing its result *is*. We therefore target calls to `ref_deref` with our analysis.

Furthermore, it is important to note that the array is never handled directly, but via a reference. Hence our analysis is concerned with the type `ref<array<'a>>`. This also means that (at least one of the components) is very similar to the reference analysis discussed earlier inReference Analysis. It follows that a look at the reference language extension is essential. Furthermore, as the array is also handled as pointer, taking a look at the pointer language extension is helpful too.

The relevant parts can be found in the header files `insieme/core/lang/reference.h` and `insieme/core/pointer.h`. First thing to note here is that the pointer language extension builds on top of the reference extension.

For the creation process of the array, `ref_decl` is used. As this operator is *not* derived, we have to handle it explicitly using an `OperatorHandler`. See the declaration in `reference.h`

```
LANG_EXT_LITERAL(RefDecl, "ref_decl", "(type<ref<'a,'c,'v,'k>>) -> ref<'a,'c,'v,'k>")
```

This size of the array (21) can be inferred from its argument `type_lit(ref<array<int<4>, 21>, f, f, plain>)`.

For the access we need to investigate the pointer language extension and immediately notice the following type alias

```
TYPE_ALIAS("ptr<'a,'c,'v>", "( ref<array<'a>,'c,'v>, int<8> )");
```

which indicates that a pointer in INSPIRE is handled as a pair of array reference and offset. Its use can be seen when inspecting the definition of the derived operator `ptr_subscript`.

```
LANG_EXT_DERIVED(PtrSubscript, R"(
    (p : ptr<'a,'c,'v>, i : int<8>) -> ref<'a,'c,'v> {
        return p.0[p.1 + i];
    }
)")
```

The offset `i`, used for the `ptr_subscript` call, is simply added to the second part of the pair `p` before the actual subscript happens.

As this function requires a pointer as argument, the original array is converted to a pointer by the frontend using the derived operator `ptr_from_array`, which in turn uses the abstract operator `ref_reinterpret`. `ref_reinterpret` is a reinterpret cast altering the actual interpretation of the referenced memory call.

```
// from pointer.h
LANG_EXT_DERIVED(PtrFromArray, R"(
    (r : ref<array<'a,'s>,'c,'v>) -> ptr<'a,'c,'v> {
        return ( ref_reinterpret(r,type_lit(array<'a,inf>)), 0l );
    }
)")

// from reference.h
LANG_EXT_LITERAL(RefReinterpret, "ref_reinterpret", "(ref<'a,'c,'v,'k>, type<'b>)"
                                                    " -> ref<'b,'c,'v,'k>")
```

The structure of Case 2 is identical to this one, the only difference is the literal for accessing the array element. We therefore now look at Case 3.

```
// case3.cpp

int main(void) {
    int a[21];
    int *ap = a + 10;
    ap[20];
}
```

```
// case3.ir

decl IMP_main : () -> int<4>;
// Inspire Program
int<4> function IMP_main (){
    var ref<array<int<4>,21>,f,f,plain> v1 =
      ref_decl(type_lit(ref<array<int<4>,21>,f,f,plain>));

    var ref<ptr<int<4>>,f,f,plain> v2 =
      ptr_add(ptr_from_array(v1), 10);

    *ptr_subscript(*v2, 20);

    return 0;
}
```

In this INSPIRE program, the array is accessed via the pointer $v2$ which is already offset by 10 elements. This offset is created by using the derived operator `ptr_add`. Its definition follows:

```
LANG_EXT_DERIVED(PtrAdd, R"(
    (p : ptr<'a,'c,'v>, i : int<8>) -> ptr<'a,'c,'v> {
        return ( p.0, p.1 + i );
    }
)")
```

As only this derived operator is used, no additional changes to the analysis are necessary compared to Case 1 and 2.

Case 4 uses a different mechanism for allocating the array.

```
// case4.cpp

int main(void) {
    auto a = new int[21];
    a[42];
    delete[] a;
}
```

```
 1  // case4.ir
 2
 3  decl IMP_main : () -> int<4>;
 4  // Inspire Program
 5  int<4> function IMP_main (){
 6      var ref<ptr<int<4>>,f,f,plain> v1 =
 7        ptr_from_array(
 8          // type
 9          <ref<array<int<4>,21>,f,f,plain>>
10
11          // memory location
12          (ref_new(type_lit(array<int<4>,21>)))
13
14          // initialisation
15          {}
16        );
17
18      *ptr_subscript(*v1, 42);
19
20      ref_delete(ptr_to_array(*v1));
21
22      return 0;
23  }
```

The syntax from line 8 to line 15 corresponds to an InitExpr, where the type is written between `< >`, the memory location between `( )`, and the initialisation between `{ }`. In this case the type is straight forward, `ref_new` is used to allocate the array on the heap, and the array is not initialised.

We therefore have to look at the operator `ref_new` and see that it is derived and uses `ref_alloc` which is an abstract operator. This abstract operator needs to be handled by the analysis using an `OperatorHandler`.

```
LANG_EXT_DERIVED(RefNew, R"(
    (t : type<'a>) -> ref<'a,f,f> {
        return ref_alloc(t, mem_loc_heap );
    }
)")

LANG_EXT_LITERAL(RefAlloc, "ref_alloc", "(type<'a>, memloc) -> ref<'a,f,f>")
```

Case 5 does not introduce any new constructs, thus we can continue our investigation process.

In Case 6, the call to `ref_deref` we are interested in is wrapped in a Declaration which in turn is wrapped in a DeclarationStmt to model the assignment. Since nothing else changed, we continue with a summary.

To summarise our findings we put together a table listing the relevant operators, shortly describing each of them.

Operator	Type	Description
`ptr_add`	*derived*	Offsets a given pointer with a given offset.
`ptr_from_array`	*derived*	Casts a given pointer to an array.
`ptr_subscript`	*derived*	Returns a reference to a specific array element.
`ref_alloc`	*abstract*	Allocate an object of the given type at a specific memory location (stack / heap), returning a reference to it.
`ref_decl`	*abstract*	Declares an object of the given type and returning a reference to it.
`ref_deref`	*abstract*	Obtain the data stored in the memory location referenced by the given reference.
`ref_new`	*derived*	Allocates an object of the given type at the heap, returning a reference to it.
`ref_reinterpret`	*abstract*	Alters the actual interpretation of the referenced memory cell.

5.3 Designing the Analysis

Our design of this new analysis is based on the investigations done in the previous section. As has already been mentioned, the type we are concerned with the most is `ref<array<'a>>`. It follows that the operators of the reference language extensions are relevant to this analysis.

We decide to split the analysis into two parts. One for identifying the index (ie offset) used to access the array, one for deriving the size (ie number of elements) of the array.

The first part can be accomplished fully by the reference analysis (see Reference Analysis). The key to this is by investigating the corresponding `DataPath` derived by the reference analysis.

For the second part we have create a new specialised DFA, tracking the number of elements of an allocated array. We will therefore refer to this analysis as *element count analysis*.

A call to our (yet to construct) function `outOfBounds` will trigger both analyses and compare the results. If an out-of-bounds access is noticed (ie an access where the offset is equal or greater to the array's element count), `IsOutOfBounds` is returned. If one (or both) of the triggered analyses could not determine an accurate enough result, `MaybeOutOfBounds` is returned. Otherwise we can safely say that no out-of-bounds (according to our set goals) occurs returning `IsNotOutOfBounds`.

Next we define the property spaces of our two *sub-analyses*.

5.4 Defining the Property Spaces

This is straightforward as we have already covered the reference analysis in Reference Analysis and `DataPath`s in Data Path Analysis. Using the established reference analysis on the offset, we will get back a set of `References`, where `References` are defined as follows:

```haskell
data Reference i = Reference { creationPoint :: Location
                             , dataPath      :: DP.DataPath i }
                 | NullReference
                 | UninitializedReference
  deriving (Eq,Ord,Show,Generic,NFData)
```

Where we are interested in the `DataPath` of a `Reference`. The type parameter `i` will be a `SimpleFieldIndex` (see [Data Path]). From the `DataPath` we can infer the index used for accessing the array.

The element count analysis provides us a `SymbolicFormulaSet` like the arithmetic analysis (see Arithmetic Analysis). This set contains `SymbolicFormula`s modelling the element count of the array. We reuse the `Lattice` and `ExtLattice` instance definitions of `SymbolicFormula`.

5.5 Defining the Interface

In this case, the interface is trivial and can be derived from the design process. The possible outcomes are gathered in a single ADT.

```haskell
OutOfBoundsResult = MayBeOutOfBounds
                  | IsNotOutOfBounds
                  | IsOutOfBounds
  deriving (Eq,Ord,Enum,Show,Generic,NFData)
```

We could have used the same approach as for the boolean analysis (seeBoolean Analysis), by defining this enumeration in a dedicated header file, which is then used in Haskell and C++ to ensure that the mapping is always correct. Yet, for simplicity, we chose not to and define an identical enumeration in C++. The definition goes into a new header file, dedicated to the out-of-bounds analysis:

```C++
// File:      insieme/analysis/cba/haskell/out_of_bounds_analysis.h
// Namespace: insieme::analysis::cba::haskell

enum class OutOfBoundsResult : int {
    MayBeOutOfBounds,
    IsNotOutOfBounds,
    IsOutOfBounds,
};
```

Furthermore we add the prototype of the analysis function to the same header file.

```C++
// File:      insieme/analysis/cba/haskell/out_of_bounds_analysis.h
// Namespace: insieme::analysis::cba::haskell

OutOfBoundsResult getOutOfBounds(Context& ctxt, const CallExprAddress& expr);
```

The Haskell function representing the out-of-bounds analysis has the following signature and goes into a new file `OutOfBounds.hs`:

```haskell
    outOfBounds :: SolverState -> NodeAddress -> (OutOfBoundsResult,SolverState)
```

As the interface for C++ as well as Haskell has now been established, the next step is to write the necessary boiler plate code in the adapter.

5.6 Setup Boilerplate

The first thing we take care of is the exporting of the Haskell `outOfBounds` function. As our result is an enumeration we can simply pass an integer representing the corresponding enum value via the Foreign Function Interface (FFI). The following code is added to `Adapter.hs`.

```haskell
foreign export ccall "hat_out_of_bounds"
  outOfBounds :: StablePtr Ctx.Context -> StablePtr Addr.NodeAddress -> IO CInt

outOfBounds ctx_hs expr_hs = do
    ctx  <- deRefStablePtr ctx_hs
    expr <- deRefStablePtr expr_hs
    let (result,ns) = OOB.outOfBounds (Ctx.getSolverState ctx) expr
    let ctx_c = Ctx.getCContext ctx
    ctx_nhs <- newStablePtr $ ctx { Ctx.getSolverState = ns }
    updateContext ctx_c ctx_nhs
    return $ fromIntegral $ fromEnum result
```

The export is available under the symbol `hat_out_of_bounds` which we import in C++ next.

```C++
extern "C" {
    namespace hat = insieme::analysis::cba::haskell;
    int hat_out_of_bounds(hat::StablePtr ctx, const hat::HaskellNodeAddress expr_hs);
}
```

This import and the wrapper around it (next code snippet) both go into a new source file `out_of_bounds_analysis.cpp`.

``{.cpp .numberLines} // Namespace: insieme::analysis::cba::haskell

OutOfBoundsResult getOutOfBounds(Context& ctxt, const core::CallExprAddress& call) { const auto& refext = call.getNodeManager() .getLangExtension();

```
if (!refext.isCallOfRefDeref(call)) {
    return OutOfBoundsResult::IsNotOutOfBounds;
}

auto call_hs = ctxt.resolveNodeAddress(call);
int res = hat_out_of_bounds(ctxt.getHaskellContext(), call_hs);
return static_cast<OutOfBoundsResult>(res);
```

} ```

As the analysis is to be only invoked on calls to the `ref_deref` operator, we import the reference language extension in lines 3 and 4 and ensure that the argument `call` is indeed a call to `ref_deref` in lines 6–8.

Note that we have to add the new Haskell module (defined by `OutOfBounds.hs`) to the list of modules in `insieme-hat.cabal`.

5.7 Write Test Cases

Before actually implementing the analysis, setting up a few test cases is a good idea. This eases the development process as we can immediately verify the correctness of our implementation. Since we have already defined six cases the analysis should cover, we can morph these cases into unit tests.

Insieme uses the Google Test Framework (aka gtest) for most of its infrastructure testing – therefore we also use gtest and integrate the tests into the infrastructure.

Important to note here is that we are not using the original input code for our tests, but the corresponding INSPIRE programs. Otherwise we would have a strong coupling between the analysis module and Insieme’s frontend.

Each test case for the out-of-bounds analysis goes into the file `out_of_bounds_analysis_test.cc` and will be structured as follows.

```C++
 1  TEST(OutOfBounds, Basic) {
 2      NodeManager mgr;
 3      IRBuilder builder(mgr);
 4      Context ctx;
 5
 6      auto stmt = builder.parseStmt(
 7          "{"
 8          "   var ref<array<int<4>, 21>> a = ref_decl(type_lit(ref<array<int<4>, 21>>));"
 9          "   *ptr_subscript(ptr_from_array(a), 42);"
10          "}"
11      ).as<CompoundStmtPtr>();
12      auto call = CompoundStmtAddress(stmt)[1].as<core::CallExprAddress>();
13
14      ASSERT_EQ(OutOfBoundsResult::IsOutOfBounds, getOutOfBounds(ctx, call));
15  }
```

In line 1, `OutOfBounds` is the group this test case belongs to and `Basic` is its (unique) name. The argument to the `parseStmt` call in line 6 is the INSPIRE program for this test. The CallExpr of interest is extracted from it in line 12 and passed to the out-of-bounds analysis as input in line 14. We expect the result to be `IsOutOfBounds`.

Note that the extracted CallExpr (input to the analysis) is a call to `ref_deref` as the problem of out-of-bound access occurs only upon dereferencing.

The file `out_of_bounds_analysis_test.cc` is placed inside the `test` sub-directory of the analysis module. Insieme’s build infrastructure will automatically create an executable allowing us to run all test cases of the out-of-bounds analysis in a single go. We are also able to run only a subset of the test cases by using the `--gtest_filter` option provided by gtest.

5.8 Implementation

The big picture of this analysis has been communicated in [Design]. Before starting with implementing the element count analysis we define ourselves the following helper functions.

```haskell
maybeToBool :: Maybe Bool -> Bool
maybeToBool = Data.Foldable.or

goesDown :: [Int] -> NodeAddress -> NodeAddress
goesDown l = foldr1 (.) $ goDown <$> reverse l
```

5.8.1 Element Count Analysis

The element count analysis is a specialised DFA and therefore requires some boilerplate code:

```haskell
data ElementCountAnalysis = ElementCountAnalysis
  deriving (Typeable)

elementCountAnalysis :: DataFlowAnalysis ElementCountAnalysis
                                          (ValueTree SimpleFieldIndex
                                          (SymbolicFormulaSet BSet.Bound10))

elementCountAnalysis = (mkDataFlowAnalysis ElementCountAnalysis "EC" elementCount)

elementCount :: NodeAddress -> Solver.TypedVar (ValueTree SimpleFieldIndex
                                                (SymbolicFormulaSet BSet.Bound10))

elementCount addr = dataflowValue addr elementCountAnalysis ops
  where -- ...
```

This analysis covers the operators of the reference language extension and therefore uses`OperatorHandler`s (`ops`) to model their semantics. The simplest case is encountered when dealing with a `ref_null` as the element count is zero.

```haskell
refNull = OperatorHandler cov dep val
  where
    cov a = isBuiltin a "ref_null"
    dep _ = []
    val a = toComposed $ BSet.singleton $ Ar.zero
```

Next, the operators `ref_cast`, `ref_reinterpret`, `ref_narrow`, and `ref_expand` simply forward the element count of their (first) argument as their semantics do not yield any modifications.

```haskell
noChange = OperatorHandler cov dep val
  where
    cov a = any (isBuiltin a) ["ref_cast", "ref_reinterpret" , "ref_narrow", "ref_expand"]
    dep _ = [Solver.toVar baseRefVar]
    val a = Solver.get a baseRefVar

    baseRefVar = elementCount $ goDown 1 $ goDown 2 addr
```

`ref_decl` and `ref_new` are used for the creation of arrays as we have observed at the beginning of this chapter. From them we can extract the size of the array.

```haskell
1  creation = OperatorHandler cov dep val
2    where
3      cov a = any (isBuiltin a) ["ref_decl", "ref_new"] && isRefArray
4      dep _ = Solver.toVar arraySize
5      val a = Solver.get a arraySize
6
7      arraySize = arithmeticValue $ goesDown [0,2,0,2,1,0] addr
8
9  isRefArray = maybeToBool $ isArrayType <$> (getReferencedType $ goDown 0 addr)
```

Note the check in line 3 as we only cover creations of arrays. From the call to`ref_decl` or `ref_new` we invoke the arithmetic analysis on the node representing the size of the array and forward its result. The location of the array size, here the node path `[0,2,0,2,1,0]` can be inferred from [Case1].

We also support scalars, their element count will be one.

```haskell
scalar = OperatorHandler cov dep val
  where
    cov a = any (isBuiltin a) ["ref_decl", "ref_new"] && not isRefArray
    dep _ = []
    val _ = toComposed $ BSet.singleton $ Ar.one
```

This completes our list of operator handlers.

```haskell
ops = [refNull, noChange, creation, scalar]
```

5.8.2 Getting the Array Index

The element count analysis provides us with a `SymbolicFormulaSet`. In order to (easily) check for out-of-bounds accesses, the array index part should provide its result in a similar data structure. From the reference analysis we get a `ReferenceSet`, where the containing `References` provide the `DataPath` from which the index can be extracted. Therefore we now construct the parts responsible for bringing the result into the wanted data structure

```haskell
toDataPath :: Ref.Reference i -> Maybe (DP.DataPath i)
toDataPath (Ref.Reference _ dp) = Just dp
toDataPath _                    = Nothing

toPath :: DP.DataPath i -> Maybe [i]
toPath dp | DP.isInvalid dp = Nothing
toPath dp                   = Just $ DP.getPath dp

toFormula :: [SimpleFieldIndex] -> Maybe SymbolicFormula
toFormula []          = Just Ar.zero
toFormula (Index i:_) = Just $ Ar.mkConst $ fromIntegral i
toFormula _           = Nothing
```

Note that each of the functions returns a `Maybe` as there exists the possibility for the conversion to fail. This would happen, for instance, if the received `ReferenceSet` contains a `NullReference`.

All that is left to dos, is to combine these parts using Kleisli composition, convert the results to`ArrayAccess`, and adjust the bound; yielding:

```haskell
data ArrayAccess = ArrayAccess SymbolicFormula
                 | InvalidArrayAccess
  deriving (Eq,Ord,Show,Generic,NFData)

convertArrayIndex :: Ref.ReferenceSet SimpleFieldIndex -> BSet.UnboundSet ArrayAccess
convertArrayIndex = BSet.changeBound
                  . BSet.map (maybe InvalidArrayAccess ArrayAccess)
                  . BSet.map (toDataPath >=> toPath >=> toFormula)
```

5.8.3 Putting it Together

Now that we have the array size and the index used to access it in a usable format, we can put the two parts together, finally implementing `outOfBounds`. `arraySize` and `arrayIndex` is processed as follows.

```haskell
(arrayIndex',ns) = Solver.resolve init
                 $ Ref.referenceValue
                 $ goDown 1 $ goDown 2 addr

arrayIndex :: BSet.UnboundSet ArrayAccess
arrayIndex = convertArrayIndex $ toValue arrayIndex'

(arraySize',ns') = Solver.resolve ns
                 $ elementCount
                 $ goDown 1 $ goDown 2 addr

arraySize :: SymbolicFormulaSet BSet.Unbound
arraySize = BSet.changeBound $ toValue arraySize'
```

First the `arrayIndex` is derived using the reference analysis and an initial `SolverState` (`init`). The resulting `SolverState` (`ns`) is then used with the new element count analysis to derive `arraySize`.

Next, each element of `arrayIndex` is compared with each element of `arraySize` and checked for out-of-bounds access.

```haskell
isOutOfBound :: ArrayAccess -> SymbolicFormula -> Bool
isOutOfBound (ArrayAccess i)    s = Ar.numCompare i s /= Ar.NumLT
isOutOfBound InvalidArrayAccess _ = True

oobs = BSet.toList $ BSet.lift2 isOutOfBound arrayIndex arraySize
```

The result of the analysis is determined by whether `oobs` contains at least a single `True`.

```haskell
outOfBounds :: SolverState -> NodeAddress -> (OutOfBoundsResult,SolverState)
outOfBounds init addr = (result,ns')
  where
    result = case () of
        _ | BSet.isUniverse arrayIndex -> MayBeOutOfBounds
          | BSet.isUniverse arraySize  -> MayBeOutOfBounds
          | or oobs                    -> IsOutOfBounds
          | otherwise                  -> IsNotOutOfBounds
```

5.9 Summary

To summarise the construction process a table is presented providing an overview of what files have been added / modified together with a short description.

File	State	Description
`Adapter.hs`	*modified*	Now also contains Haskell export declarations for analysis function.
`OutOfBounds.hs`	*added*	Defines the out-of-bounds analysis and its property space.
`insieme-hat.cabal`	*modified*	Contains a list of all Haskell modules of .
`out_of_bounds_analysis.cpp`	*added*	Wraps the imported out-of-bounds analysis function for C++.
`out_of_bounds_analysis.h`	*added*	Defines the analysis result and function prototype in C++.
`out_of_bounds_analysis_test.cc`	*added*	Contains a unit test suite for the out-of-bounds analysis.

The adapter and `insieme-hat.cabal` are the only files that have been modified, all other files are simply added to the code base. Note that it is not mandatory to have all FFI export declarations located in the adapter. We could also have stated those directives in `OutOfBounds.hs`, yet we decided against it in order to keep all FFI export declarations (and their relevant logic) in one single (Haskell) module.

6 Evaluation

The evaluation of the toolkit is done by using the newly constructed analysis from Constructing a new Analysis. This process is split into two parts, one focusing on the *qualitative* features, the other focusing on the *quantitative*.

6.1 Qualitative Evaluation

The qualitative evaluation focuses on the ability of the toolkit to lift an analysis to more complex input programs than explicitly covered by the analysis implementation. Non of the explicitly developed analyses components is dealing with function calls. Yet, as we have used the mechanisms provided by the framework, no modifications are needed to make the analysis inter-procedural. To show this, two additional test cases are constructed:

```
 1  // case7.ir
 2
 3  def fun = function (arr : ref<ptr<int<4>>>) -> ptr<int<4>> {
 4      return ptr_add(*arr, 20);
 5  };
 6
 7  {
 8      var ref<array<int<4>,21>> arr = ref_decl(type_lit(ref<array<int<4>,21>>));
 9      var ref<ptr<int<4>>> ap = fun(ptr_from_array(arr));
10      *ptr_subscript(*ap, 10);
11  }
```

```
 1  // case8.ir
 2
 3  def fun = function (arr : ref<ptr<int<4>>>) -> ptr<int<4>> {
 4      return ptr_add(*arr, 10);
 5  };
 6
 7  {
 8      var ref<array<int<4>,21>> arr = ref_decl(type_lit(ref<array<int<4>,21>>));
 9      var ref<ptr<int<4>>> ap = fun(ptr_from_array(arr));
10      *ptr_subscript(*ap, 10);
11  }
```

Note the different offset in line 2 (both cases).

For the `ref_deref` in line 8 (the first `*` operator in both cases), `IsOutOfBounds` must be returned by the analysis in Case 7, and `IsNotOutOfBounds` in Case 8.

We also provide two more test cases running the analysis on a scalar, instead of an array.

```
1  // case9.ir
2
3  {
4      var ref<int<4>> a = ref_decl(type_lit(ref<int<4>>));
5      var ref<ptr<int<4>>> ap = ptr_from_ref(a);
6      *ptr_subscript(*ap, 42);
7  }
```

```
1  // case10.ir
2
3  {
4      var ref<int<4>> a = ref_decl(type_lit(ref<int<4>>));
5      var ref<ptr<int<4>>> ap = ptr_from_ref(a);
6      *ptr_subscript(*ap, 0);
7  }
```

Case 9 should yield `IsOutOfBounds` while Case 10 should yield `IsNotOutOfBounds`.

As with the initial six cases in [Tests], Cases 7–10 are converted to unit tests and executed.

```
$ ./ut_analysis_cba_haskell_out_of_bounds_analysis_test --gtest_filter=*AcrossFunctionCall*
Running main() from gtest_main.cc
Note: Google Test filter = *AcrossFunctionCall*
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from OutOfBounds
[ RUN      ] OutOfBounds.AcrossFunctionCall
[       OK ] OutOfBounds.AcrossFunctionCall (472 ms)
[ RUN      ] OutOfBounds.AcrossFunctionCallNotOutOfBounds
[       OK ] OutOfBounds.AcrossFunctionCallNotOutOfBounds (286 ms)
[----------] 2 tests from OutOfBounds (758 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (759 ms total)
[  PASSED  ] 2 tests.

$ ./ut_analysis_cba_haskell_out_of_bounds_analysis_test --gtest_filter=*Scalar*
Running main() from gtest_main.cc
Note: Google Test filter = *Scalar*
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from OutOfBounds
[ RUN      ] OutOfBounds.Scalar
[       OK ] OutOfBounds.Scalar (456 ms)
[ RUN      ] OutOfBounds.ScalarNotOutOfBounds
[       OK ] OutOfBounds.ScalarNotOutOfBounds (281 ms)
[----------] 2 tests from OutOfBounds (737 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (737 ms total)
[  PASSED  ] 2 tests.
```

The new test cases pass, evincing that the framework indeed transparently integrates support for language features not explicitly covered.

6.2 Quantitative Evaluation

For the quantitative evaluation we utilise Insieme's integration tests and run the out-of-bounds analysis on every`ref_deref` contained. For this, we do not care about the actual result, but about the runtime. This evaluates whether the analysis and framework provide sufficient enough performance.

As the integration tests are input files to the compiler, they are only available as C/C++ sources. We therefore chose to create another driver (`haskell_dumper`) dedicated to reading C/C++ input files and dumping the binary representation of the corresponding INSPIRE program. Next, this dump is read by another binary (`insieme-hat`) which then collects all node address of `ref_deref` calls and runs the out-of-bounds analysis on them. To maximise modularity, the part for finding target nodes (in this case calls to `ref_deref`) and running the analysis is kept concise in dedicated functions.

```haskell
findTargets :: NodeAddress -> [NodeAddress] -> [NodeAddress]
findTargets addr xs = case getNode addr of
    IR.Node IR.CallExpr _ | Q.isBuiltin (goDown 1 addr) "ref_deref" -> addr : xs
    _ -> xs

analysis :: NodeAddress -> State SolverState OutOfBoundsResult
analysis addr = do
    state <- get
    let (res, state') = outOfBounds state addr
    put state'
    return res
```

The binary dump is read from `stdin` and each analysis run uses the `SolverState` outputted by the previous run.

```haskell
 1  main :: IO ()
 2  main = do
 3      -- parse binary dump (valid input expected)
 4      dump <- .getContents
 5      let Right ir = parseBinaryDump dump
 6
 7      let targets = foldTreePrune findTargets Q.isType ir
 8      let res = evalState (sequence $ analysis <$> targets) initState
 9
10      start <- getCurrentTime
11      evaluate $ force res
12      end   <- getCurrentTime
13
14      let ms = round $ (*1000) $ toRational $ diffUTCTime end start :: Integer
15      printf "%d ms" ms
```

Note that in line 11 `res` is evaluated to normal form using `force` from the `deepseq` package. This is necessary since Haskell features lazy-evaluation and `evaluate` will only evaluate the argument to weak head normal form[14] (ie not fully). `force` on the other hand evaluates the argument completely.

[14] See https://wiki.haskell.org/Weak_head_normal_form for more details.

The evaluation is performed on an `Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz` with 256 GB of memory.

Overall 320 integration tests have been used, containing about 3.81 x10^8 nodes (not shared) altogether, from which 63 662 are calls to `ref_deref`. To minimise measurement errors, each integration test is processed by `insieme-hat` 10 times from which the arithmetic mean of the outputted execution times is recorded. The result is displayed in Figure 11. It shows that the majority of the test cases can be completed in under 30 ms. The full list of integration tests (with the average execution time of the 10 runs) is provided in [Integration Tests].

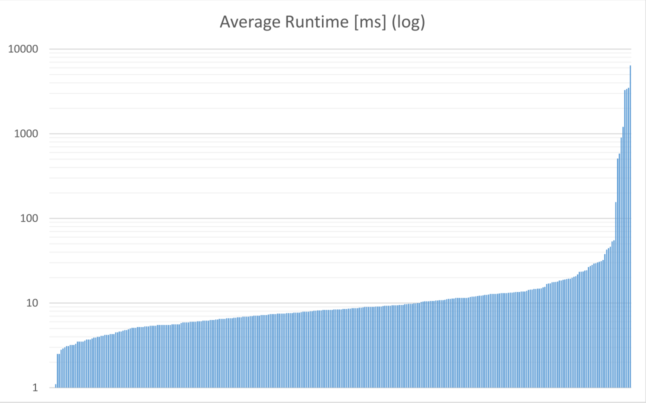


Figure 11: Runtime of the 320 test cases.

Furthermore, memory consumption has been recorded using `GNU time 1.7`. Figure 12 displays the recorded `Maximum Resident Set Size`. The majority of test cases do not consume more than 100 MB of memory.

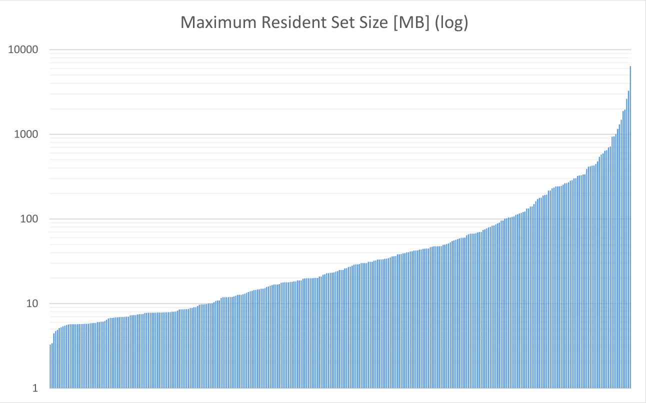


Figure 12: Memory consumption of the 320 test cases.

7 Conclusion

In this thesis a framework for implementing various static program analyses has been presented. It utilises a Constraint-Based Analysis (CBA) framework, which is originally derived from the Data-Flow Analysis (DFA) framework presented by Nielson, Nielson, and Hankin (1999) and has further been customised by Jordan (2014). This toolkit, created using Haskell, integrates seamlessly into the Insieme compiler, but can also be used as standalone.

It enables developers and researchers to rapidly develop and use specialised static program analyses, for various different kinds of applications (eg auto-tuning or identifying optimisation candidates in a compilation process). In Constructing a new Analysis we have shown that crafting a new analysis only requires a small amount of boiler plate code and some moderate implementation effort. Furthermore we have shown that the framework elevates the capabilities of newly realised analyses while maintaining decent performance.

7.1 Future work

While the framework's current state already allows one to use it for various tasks, certain language features of the INSieme Parallel Intermediate REpresentation (INSPIRE) are not yet supported. As stated in Objectives, this merely builds the foundation upon which development will iterate on in future.

Regarding documentation, the core concept of the framework itself, as well as analyses implemented using it, are quite straight forward to grasp. Yet, further documentation of certain edge cases would be helpful. However, the main challenge encountered by new developers is understanding the different node types, constructs, and language extensions of INSPIRE. This issue can be addressed by providing more in-depth documentation on language extensions and their underlying semantics, as well as their connections to other language extensions and the INSPIRE core language.

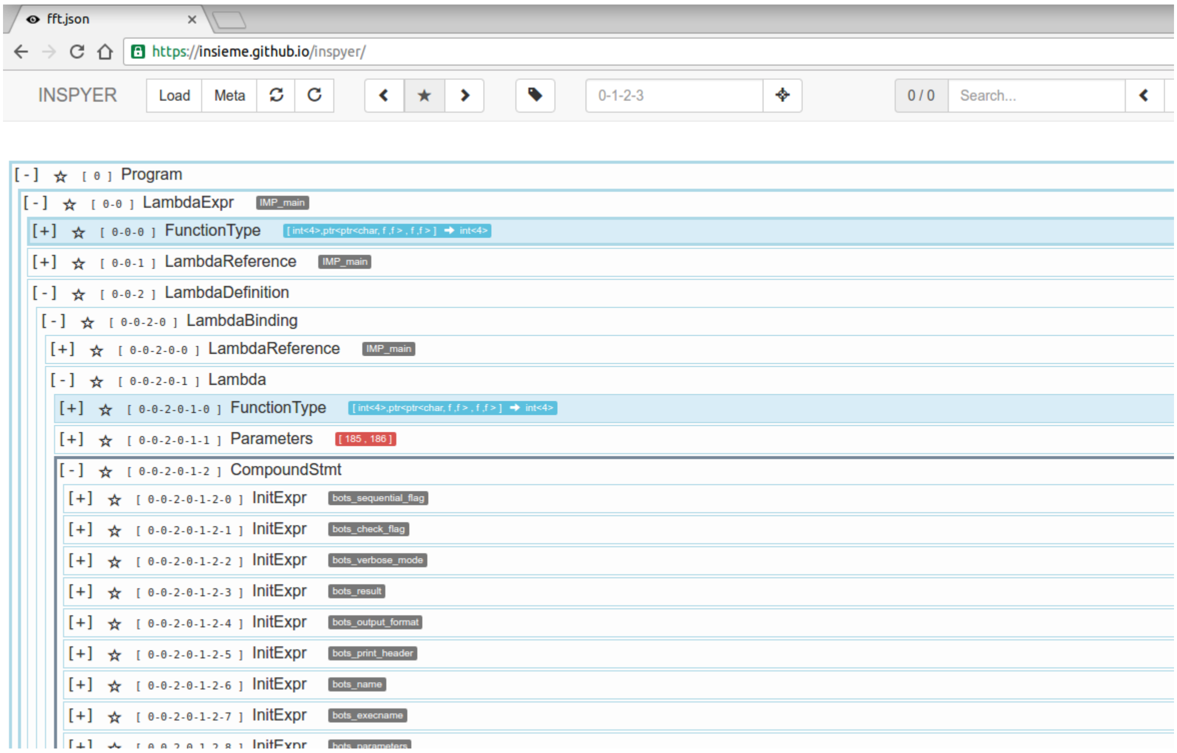
On the framework side, creating a catalogue of convenience functions (mainly queries) for all the different INSPIRE constructs would further ease the creation and prototype process of analysis.

Unit-testing an analysis is not that much of a hassle when using the framework together with Insieme. For standalone purposes, however more tooling would be required to be efficient at writing tests.

And last, but certainly not least, debugging techniques and tools are needed. At the current state of development, debugging mainly resorts to using Haskell's `Debug.Trace` and dumping the computed Assignment, render it via Graphviz. A notable improvement would be to have a dedicated viewer for the Assignment, in a similar fashion as INSPYER to INSPIRE.

8 Appendix A: INSPYER

The INSPYER tool was created to visualise a program represented in Insieme's Intermediate Representation (IR), also known as the INSieme Parallel Intermediate REpresentation (INSPIRE), in an interactive way. Using the text representation of INSPIRE, as seen in Semantics, is helpful for inspecting an INSPIRE program. Yet it is a bit cumbersome to work with – even when using an editor, like Vim, which supports code folding, jumping to corresponding closing brackets, and allows bookmarking locations. The main idea is to have a Graphical User Interface (GUI) application which allows one to collapse subtrees which are not interesting and display useful information for each node in one line (ie row). Navigation functions like *goto* and *search* are available as well.



```
[-] ☆ [ 0 ] Program
 [-] ☆ [ 0-0 ] LambdaExpr  IMP_main
  [+] ☆ [ 0-0-0 ] FunctionType  [ int<4>,ptr<ptr<char, f ,f > , f ,f > ] ➔ int<4>
  [+] ☆ [ 0-0-1 ] LambdaReference  IMP_main
  [-] ☆ [ 0-0-2 ] LambdaDefinition
   [-] ☆ [ 0-0-2-0 ] LambdaBinding
    [+] ☆ [ 0-0-2-0-0 ] LambdaReference  IMP_main
    [-] ☆ [ 0-0-2-0-1 ] Lambda
     [+] ☆ [ 0-0-2-0-1-0 ] FunctionType  [ int<4>,ptr<ptr<char, f ,f > , f ,f > ] ➔ int<4>
     [+] ☆ [ 0-0-2-0-1-1 ] Parameters  [ 185 , 186 ]
     [-] ☆ [ 0-0-2-0-1-2 ] CompoundStmt
      [+] ☆ [ 0-0-2-0-1-2-0 ] InitExpr  bots_sequential_flag
      [+] ☆ [ 0-0-2-0-1-2-1 ] InitExpr  bots_check_flag
      [+] ☆ [ 0-0-2-0-1-2-2 ] InitExpr  bots_verbose_mode
      [+] ☆ [ 0-0-2-0-1-2-3 ] InitExpr  bots_result
      [+] ☆ [ 0-0-2-0-1-2-4 ] InitExpr  bots_output_format
      [+] ☆ [ 0-0-2-0-1-2-5 ] InitExpr  bots_print_header
      [+] ☆ [ 0-0-2-0-1-2-6 ] InitExpr  bots_name
      [+] ☆ [ 0-0-2-0-1-2-7 ] InitExpr  bots_execname
      [+] ☆ [ 0-0-2-0-1-2-8 ] InitExpr  bots_parameters
```

Figure 13: Screenshot of INSPYER visualising the FFT integration test.

INSPYER is created as server-less web application using only HTML[15], CSS[16], and JavaScript. This allows for it to be run locally using a browser or even host it online – in our case we are using GitHub Pages. Bootstrap, an HTML, CSS, and JavaScript framework is used to create the user interface in little time and to provide a pleasing look and feel. A screenshot of INSPYER can be seen in Figure 13 where Insieme's FFT[17] integration test has been loaded.

[15] HyperText Markup Language

[16] Cascading Style Sheets

[17] Fast Fourier Transformation

8.1 JSON Export

An INSPIRE program is exported using Insieme's *JavaScript Object Notation (JSON) dumper*. The format is straight forward and similar to the binary dump. An example is given in the next code snippet. The JSON object is required to have a field named `root` which identifies the root node. Each Intermediate Representation (IR) node is attached to the object using a unique key, for simplicity the original memory address is used. The information dumped for one node consists of the node type, its children and (in case its a value node) the value. One can already see in the example, that the list of children is just a list of the memory locations of the child nodes. The memory location serves as a unique identifier for each node. This representation produces node sharing, similar to the binary dump.

```
{
    "root" : "0x30b8270",
    "0x30b8270" : {
        "Kind" : "Program",
        "Children" : ["0x30b86c0"]
    },
    "0x30b86c0" : {
        "Kind" : "LambdaExpr",
        "Children" : ["0x218f910","0x21ad1b0","0x30b8760"]
    },
    "0x1f588f0" : {
        "Kind" : "StringValue",
        "Value" : "int",
        "Children" : []
    },
    ...
}
```

The user loads the JSON dump either by selecting it via the corresponding dialogue (using the button labelled*Load* located in the top bar) or by dragging and dropping the file into the main area of INSPYER. Doing this will trigger the JavaScript logic, loading the file using `FileReader` and `JSON.parse`. The JSON dump is then interpreted. A class `Node`, dedicated for maintaining and working with nodes, has been implemented. The tree visualisation is generated incrementally – upon expanding a node, its children will be created. This keeps the DOM[18] small and simple, which is important to provide adequate performance – even small input programs can yield a big INSPIRE program. Having more than 1 000 nodes displayed simultaneously results in an unresponsive (ie laggy) user experience. This limitation is inherited by the browser.

[18] Document Object Model

8.2 User Interface

The tree can be navigated either by using the mouse or the keyboard. Clicking on a node will cause it to expand (or collapse if it is already expanded). When using the keyboard, the currently *selected* node is outlined by a thicker, darker border. Hitting `Space` will do the same to the selected node as clicking on it. Other nodes can be selected by using the cursor keys to move the selection.
↑ and ↓ will change the selection on the current level of the tree.← moves the selection to the parent node. → selects the first child of the currently selected node, expanding it if necessary.

Each line, representing a node, consists of (from left to right):

- an indicator showing whether the node is expanded or collapsed;
- a bookmark button;
- the corresponding node address;
- the node type;
- its INSPIRE type* (eg `int<4>`);
- its value* (eg `9` or `"Hello"`;
- its variable id*; and
- the unique identifier in form of the original memory location.

* if available

The controls (navigation bar in Figure 13) of INSPYER allow you to (again from left to right):

- load an INSPIRE program from a JSON dump;
- load meta information from a *meta file* (see Meta Data);
- reload the JSON dump and meta file (keeping bookmarks and search results);
- reload the JSON dump and meta file (clearing bookmarks and search results);
- move the selection back and forth between bookmarked nodes;
- show / hide node addresses – useful deep inside the tree, where node addresses can get unpleasantly long;
- directly jump to a node given a node address;
- search the tree using a REGular EXpression (REGEX), jumping back and forth between results; and
- show a modal window containing information about hotkeys.

The modal window can be extended easily to hold more, infrequently used functions. There are currently no such functions implemented, but may be needed in the future. An example for this would be statistics about the currently loaded dump.

The search functionality is implemented using web workers. This allows us to search the tree using a dedicated thread – the GUI remains responsive even if it takes multiple seconds to search the whole tree. The set of search results will be updated periodically so the user can inspect results matching the query before the search process is completed.

The FFT example contains 34 254 nodes and can be fully searched in about 15 seconds – 'InitExpr' was used as search string, yielding 25 hits. Note that the number of hits has a huge performance impact as each hit results in a new JavaScript object, which also has to be transferred from the web worker to the main thread. Searching the FFT example for 'StringValue' yields over 2 500 000 results before the browser kills the main thread (and web worker). This took about two minutes to happen. These tests have been run on an `Intel(R) Core(TM) i3-3120M CPU @ 2.50GHz` with `16 GB DDR3 Memory @ 1600Hz` in `Chromium 51`.

The web worker API requires an external script file for the thread, but loading the script later on is not possible from a clients filesystem due to security concerns. A workaround is to have the script embedded into the page and create a Binary Long OBject (BLOB) from it. This BLOB will then be passed to the worker. Therefore the logic of the search web worker is found in `index.html` at the bottom, inside a script tag with type `javascript/worker`. It is important to set the type to something different from `text/javascript` to prevent the browser from executing this code right away.

8.3 Meta Data

The *meta file* is a JSON formatted file that is associated with one JSON dump of an INSPIRE program and holds additional information. Neither what kind of information can be contained nor where that information comes from is fixed. The meta file can:

- bookmark specific nodes, the first bookmark will be jumped to on load;
- expand specific nodes;
- highlight specific nodes;
- attach labels to specific nodes; and
- attach multiline information (including markup), shown when a node is expanded.

As can be seen in the next code snippet, node addresses are used to reference the specific nodes.

```
{
  "bookmarks": [
    "0-0-2-0-1-1",
    "0-0-2-0-1-2"
  ],
  "expands": [
    "0-0-2-0-1-2-1-0",
    "0-0-2-0-1-2-6-0",
    "0-0-2-0-1-2-12-0",
    "0-0-2-0-1-2-15-0"
  ],
  "labels": {
    "0-0-2-0-1-2-1": "some information",
    "0-0-2-0-1-2-3": "some other information"
  },
  "highlights": [
    "0-0-2-0-1-2-4",
    "0-0-2-0-1-2-5",
    "0-0-2-0-1-2-6",
    "0-0-2-0-1-2-7",
    "0-0-2-0-1-2-8"
  ],
  "bodies": {
    "0-0-2-0-1-2-6": "Some <b>additional</b> information text"
  }
}
```

Insieme maintains a helper class for generating meta files. It is currently used in the analysis stress test and the module responsible for semantic checks. The class declaration is displayed in the next code snippet. One can manually instantiate this class, save the information and export it afterwards.

```C++
class MetaGenerator {
  private:
    NodePtr root;
    std::set<NodeAddress> bookmarks;
    std::set<NodeAddress> expands;
    std::set<NodeAddress> highlights;
    std::map<NodeAddress, std::string> labels;
    std::map<NodeAddress, std::string> bodies;

    void checkRoot(const NodePtr root);

  public:
    explicit MetaGenerator(const NodePtr root);
    void addBookmark(const NodeAddress addr);
    void addExpand(const NodeAddress addr);
    void addHighlight(const NodeAddress addr);
    void addLabel(const NodeAddress addr, const std::string label);
    void addBody(const NodeAddress addr, const std::string body);
    void dump(std::ostream& out);
};
```

This, on the other hand, requires making the instance available at every point where information needs to be added – and at the location where it should be exported. Because of this, and since there usually only exists one INSPIRE program per run of the Insieme compiler, one static instance of the class and functions wrapping the default instance’s functionality are provided. The next code snippet shows these provided utility functions.

```C++
void addBookmark(const NodeAddress addr);

void addExpand(const NodeAddress addr);

void addHighlight(const NodeAddress addr);

void addLabel(const NodeAddress addr, const std::string label);

void addBody(const NodeAddress addr, const std::string body);

void dumpTree(std::ostream& out, const NodePtr root);

void dumpMeta(std::ostream& out);
```

9 Appendix B: Evaluation Data

The attached file contains the data of all 320 integration test runs used for the quantitative evaluation. Each test has been run 10 times and the execution time of the out-of-bounds analysis, as well as the overall memory consumption has been recorded. The data provided shows the average time consumed by all 10 runs.

🗎 Quantitative Evaluation Recorded Data

References

Chambers, Craig. 2006. "Slides: Lattice-Theory Data Flow Analysis." https://courses.cs.washington.edu/courses/cse501/06wi/slides/01-18.slides.pdf.

Chong, Stephen. 2011. "Slides: Dataflow Analysis." http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf.

Insieme Team. 2017. "Insieme Compiler Project." http://insieme-compiler.org.

Jones, Mark P. 2000. "Type Classes with Functional Dependencies." In *European Symposium on Programming*, 230–44. Springer.

Jordan, Herbert. 2014. "Insieme — a Compiler Infrastructure for Parallel Programs." PhD thesis, University of Innsbruck. http://www.dps.uibk.ac.at/~csaf7445/pub/phd_thesis_jordan.pdf.

Jordan, Herbert, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. "INSPIRE: The Insieme Parallel Intermediate Representation." In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 7–18. IEEE Press.

Jovanovic, Nenad. 2005. "Lattice Tutorial." http://old.iseclab.org/people/enji/infosys/lattice_tutorial.pdf.

Lemmer, Ryan. 2015. *Haskell Design Patterns*. Packt Publishing Ltd.

Lipovaca, Miran. 2011. *Learn You a Haskell for Great Good!: A Beginner's Guide*. 1st ed. San Francisco, CA, USA: No Starch Press. http://learnyouahaskell.com.

Nielson, Flemming, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Nordenberg, Jesper. 2012. "My Take on Haskell Vs Scala." http://jnordenberg.blogspot.co.at/2012/05/my-take-on-haskell-vs-scala.html.

O'Sullivan, Bryan, John Goerzen, and Don Stewart. 2008. *Real World Haskell*. 1st ed. O'Reilly Media, Inc.

Strout, Michelle. 2006. "Slides: Lattice Theoretic Framework for DFA." http://www.cs.colostate.edu/~mstrout/CS553Fall06/slides/lecture10-lattice.

Thomasson, Samuli. 2016. *Haskell High Performance Programming*. Packt Publishing Ltd.

Wikipedia. 2016a. "Lattice (Order)." https://en.wikipedia.org/w/index.php?title=Lattice_(order)&oldid=754335711.

———. 2016b. "Reaching Definition." https://en.wikipedia.org/w/index.php?title=Reaching_definition&oldid=723830514.

———. 2017a. "Control Flow Graph." https://en.wikipedia.org/w/index.php?title=Control_flow_graph&oldid=758176946.

———. 2017b. "Hasse Diagram." https://en.wikipedia.org/w/index.php?title=Hasse_diagram&oldid=757730257.