M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Meta specifications of program analyses

Master Thesis

Mike Castro Lundin (s162901)

Kongens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Learning and understanding static analyses using the monotone framework can be a complicated and time consuming task. To simplify the process, tools such as the ones provided in `formalmethods.dk` have been developed for people to check their understanding and easily experiment with concepts.

Equivalently, learning to formulate and implement own custom analyses within the monotone framework can also challenging. The user must ensure that transfer functions are monotone, and specify a domain that is a complete lattice that satisfies the ascending and descending chain conditions.

This thesis creates and implements a tool for automating part of the process of creating a custom analysis, and thus provides a sandbox environment for quick implementation and testing of analyses, as well as comparing analyses in isomorphic or similar domains.

This project evaluates and implements extended capabilities for the monotone framework. Specifically, the developed tool automatically generates an F# data type for the domain and the F# functions for operations needed by the monotone framework to calculate the analysis result. The domain also automatically gets the bottom and top elements when they can be inferred directly from the domain. Another capability offered by the tool is automatic checking if transfer functions are monotone by the use of QuickChecking.

Preface

This master thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a masters degree in Computer Science and Engineering.

Kongens Lyngby, June 22, 2018

Mike Castro Lundin (s162901)

Acknowledgements

I would like to thank my supervisor, Professor Hanne Riis Nielson for the help, guidance and time given during my master thesis. I would also like to thank friends with whom i have had informative conversations, and have discussed and evaluated diverse approaches to take during the development of this thesis. As always I am thankful to my family.

Contents

CHAPTER 1

Introduction

Static analysis is a methodology which is used to study programs, and infer properties in them without execution. It is a multipurpose tool that is used within many contexts in software, including but not limited to security, correctness and optimisation. As time has passed, compilers have adopted a more active role in testing, debugging, development and optimisation by analysing code and thus facilitating the software development process.

Given that static analysis of programs is such a generic concept which can be applied to many contexts, the notion of the monotone framework established a structure (analysis domains and functions) in which to create analyses that could be solved by the same generic algorithms. It is essential to specify language-independent abstract interpretations in order to obtain reusable implementations or at least to minimise the language-dependent part [CC95].

The monotone framework defines a set of demands for the analysis domain and functions so that the analysis will terminate, and the algorithm will return the optimal solution within the analysis specification. Furthermore it allows to detect and group analyses in regard to their efficiency.

1.1 Objective and scope

The goal of this project is to create a design tool in which an instance of a monotone framework may be specified. Some properties of monotone framework will be verified, and some of the code for running the analysis will be generated automatically. The purpose of this tool is to simplify the process of creating new analyses, as well as comparing alternative formulations of the same analysis, and making static analysis using Abstract Interpretation more accessible.

This project will specify a programming language syntax to accept and analyse. The specification of the analysis will be also be defined, including the domain and function specifications, as well as the context in which it will be tested. The specification will be checked, within some limitations, such that the domain and transfer functions are compliant with the specification of the monotone framework. Some of these guarantees will be done by only accepting inputs which are proven to be compliant, and

some will be tested using QuickChecking to try and find a counter-example.

The user of the tool must have a basic understanding of how F# data structures such as tuples, sets, maps and discriminated unions are used, and be able to write functions that comply with a given signature. The user must also be able to read and use grammars specified in Backus–Naur form (BNF). Finally, when using the more advanced features, such as *Graph Domain*s, the user must have an understanding of how to define an ordering for a lattice in such a way that it is a complete lattice. While knowledge of the monotone framework is technically not necessary, it will be helpful when defining analyses at least on a conceptual level, and understanding the limitations of the monotone framework.

As previously mentioned, the great advantage of the monotone framework is its language independent nature. It is a goal to keep the developed framework as language-independent as possible, and to allow users to use their own language specifications (by extension lexers/parsers) in the developed tool.

1.2 Literature and tool review

Static analysis has become an increasingly large area in computer science. There are large amounts of research, commercial and open-source projects that work within the field in varying contexts, and providing a variety of capabilities. There are several ways of classifying projects into similar groups, and even when using the same criteria, scientific papers and documentation tend to use different terminology. This paper will use the terminology used by the documentation of *AbsInt* [FG13], which is one of the companies providing products withing static analysis.

Static analyses are classified into:

Syntax-based: These products offer capabilities such as style checking to comply with diverse standards. This may be something as simple as demanding that C code does not use unsafe functions such as *gets*, or verifying that variables follow certain patterns in their identifiers. The technical capabilities can vary from simple string search checks, to rules that require knowledge of the syntax.

Unsound semantics-based: These product offer capabilities such as bug finding, or vulnerability finding. Some examples of this may be static analysis to detect potential SQL injection (SQLi), Cross-site scripting (XSS) and backdoors. These analyses can detect some instances of these undesired events, but do not prove that no bug or vulnerability exists.

Sound semantics-based/Abstract Interpretation-based: These products can give guarantees that all bugs targeted by the analysis are found. This is because these products perform over approximation. As a consequence this can lead to varying amounts of false positives and warnings of limited relevance. Some examples of products in this class are *Mathworks Polyspace* and *Astrée*.

It is not unusual that products offer capabilities within more than one category, for example *Astrée* provides both Abstract Interpretation-based static analysis, as well as syntax-based static analysis capabilities.

The vast majority of products offer a black box solution, where the software runs a set of standard static analyses, and does not offer any customisation. These products are irrelevant to this project, since the focus of this framework is the customisation and design of custom analysis for the context.

There are four ways that products allow their users to customise their products to fit any custom static analysis needs:

- User rule specification: Products often limit user customisation to syntax-based analyses. This is sometimes done by allowing the user to specify some functions that verify the generated abstract syntax trees (ASTs), or sometimes use some custom defined language with limited expressivity (such as "Global variables MUST USE names matching g_" in [GRA14]). Some examples of programs that use this kind of customisation are *Constantine* [GRA14], *Microsoft Roslyn* analysers, and *Astrée*.
- Assertion style analyses: Products offer ways for users to specify in the code certain checks that must be held. This can be done for example using inline code:

```
1    void foo(int x) {
2    #ifdef __CSURF__
3        csonar_trigger(x, "==", -1, "Dangerous call to foo()");
4    #endif __CSURF__
5    ...
6    }
```

Listing 1.1: Inline code as used in [And]

or using a replacement function

```
1    void csonar_replace_foo(int x) {
2        csonar_trigger(x, "==", -1, "Dangerous call to foo()");
3        foo();
4    }
```

Listing 1.2: Replacement function as used in [And]

Some examples of programs which use this kind of customisation are *Astrée* and *GrammaTech CodeSonar*.

- API: Products offer the user access to the analysis internals using an API. This may mean that the user can write analyses in a plugin, or allow the user full access to the analysis process by allowing custom functions to run on nodes. Some examples of programs are *Clang Static Analyzer* which allows the user full access to the analysis process, *IKOS* that uses plugins [Bra+14]; and *GrammaTech CodeSonar* which offers an API.
- Domain customisation: There is also a tool which allows the user to customise the used domain to some extent (limited to octagon domains used by ELINA, polka domain used by APRON, etc). This is allowed by *crab*.

It is also worth noting that a tool named *Phasar* is currently in development which seems to have a similar purpose as the framework developed in this project. "Phasar has been developed with the goal to make static analysis easier, more accessible. Furthermore, it tries to establish a novel platform to evaluate new concepts and ideas in the area of program analysis." [Sch18] The project has been in development for over a year, and there is currently no official documentation on it's capabilities.

It is worth noting that most Abstract Interpretation-based tools offer analysis of either the LLVM intermediate representation, or C/C++. Many products offer interval analysis, dead code detection and various forms of pointer analysis. Interprocedural analysis is sometimes described as an advanced feature in some products.

In comparison to other existing products, the developed framework is an Abstract Interpretation-based static analyser, that allows customisation in what is closest to the API method. The developed framework also makes Abstract Interpretation-based static analysis even more accessible, as it does a lot of the work for the user, by generating code such that the monotone framework condition of the domain being a complete lattice is kept, and by using QuickChecking to verify the monotonicity of the specified transfer functions. This makes it a novel tool for quick design and test of analysis domains and transfer functions, and get an indication that the specified transfer functions are indeed monotonic, without manual testing, or knowledge of QuickChecking.

1.3 Overview

The developed tool will work within the monotone framework, which can be seen in Figure 1.1.

The orange blocks and arrows of the diagram are language dependent, and are specifying the language so it may be parsed, transforming it into an abstract syntax tree,

evaluating the flow of the program and having transfer functions to match statements.

The black blocks and arrows of the diagram are language independent parts of the monotone framework, and include setting the constraints that are to be solved in respect to the program graph and the worklist algorithm.

The red blocks and arrows specify what the analysis is doing, by using a complete lattice domain, a set of transfer functions and the analysis specification (if the analysis requires the greatest or least sets to solve the equations, and the direction of edge evaluation).

The input to the monotone framework is code in the language that is being analysed. User inputs are shown in blue.

The main output of the framework is the analysis results, but can also include information about the syntax/semantics of the input code or performance information about the worklist algorithm. Framework outputs are shown in green.

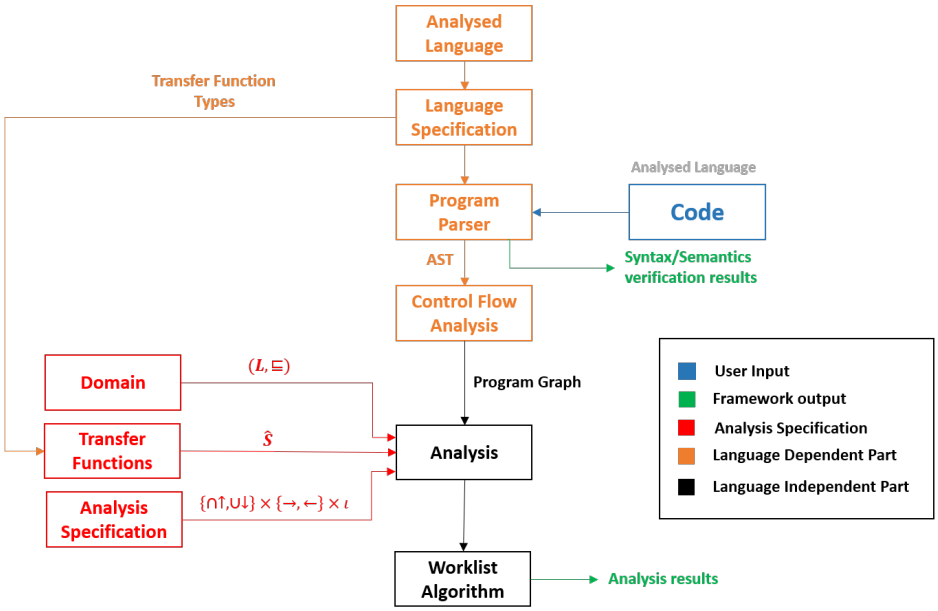


Figure 1.1: The monotone framework

This project aims to develop an extended version of the monotone framework, which can be seen in Figure 1.2. A more detailed description of the extended framework can be found in Section 6.1. This new version will extend the capabilities of the monotone framework to generate code for domain specified in the MetaL language, which

is defined by this project. Specifically a data type, ordering relations, combination operators and bottom/top elements for the lattice will be generated.

The idea is to simplify the process, by ensuring in a framework level that the domain being used is a complete lattice, and that no matter the analysis, the domain will always satisfy the requirements set by the monotone framework. Having automatically generated code also reduces the possible places where a user might make a mistake, and specify for example a preordered set.

The extended monotone framework will also help the user test the monotonicity of the transfer functions, by generating code that will use QuickChecking to test the transfer functions with randomly generated inputs.

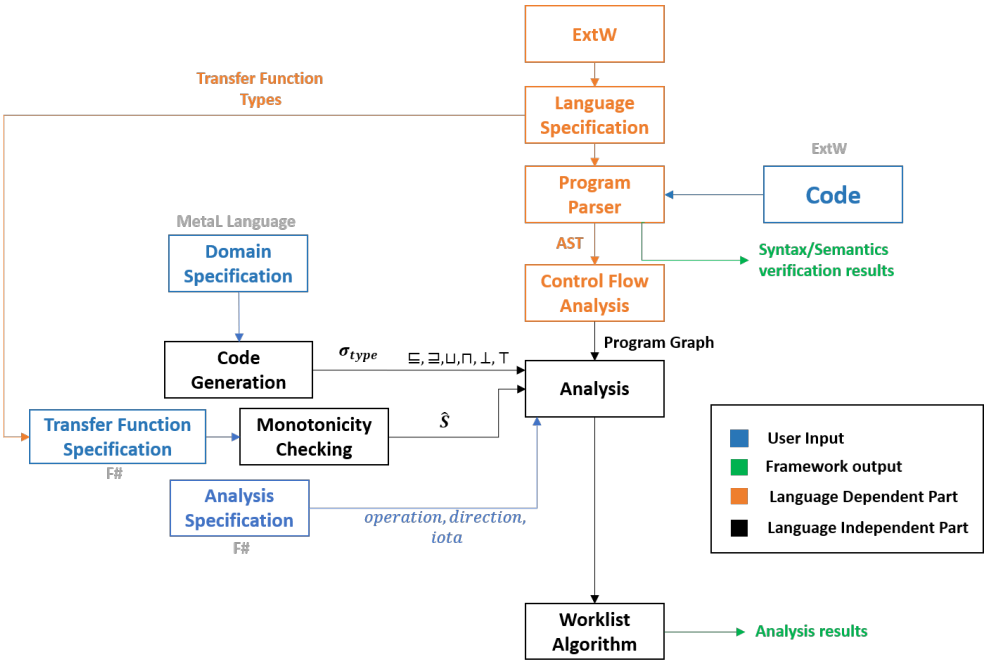


Figure 1.2: The extended monotone framework

The project implementation can be found in Github (`https://github.com/mikecl24/ProgramAnalysisFramework`), and instructions on how to run the framework can be found in Appendix A.

CHAPTER 2

ExtWhile Specification

The choice of language to analyse is mostly trivial in terms of the framework. No matter the target language, the specific code must be transformed into a program graph. Program analysis can then be done on the graphs using the constraint solving algorithm proposed by the monotone framework.

Thus in principle any language can be used in the framework, provided that there is a module to transform the code into program graphs (called pre-processing module in the implementation).

For demonstration purposes a self-defined language, which is an extension and concretization of the abstract syntax While proposed in [NNH15] is defined in Section 2.1, and the definition of program graphs in the framework, as well as algorithms to transform code into program graphs is described in Section 2.2.

There are three major side effects to the language specification and transformation into graphs being used. The first effect is in regards to analysis domains. For example if the graph transformation allows multidigraphs/quivers to be generated as program graphs, then it wont be possible to distinguish between nodes by knowing the start and end nodes only.

The second effect is in regard to the transfer function specification. It is important that the specification contains a function for each of the possible actions that can be in an edge.

The third effect is in regards to matching an edge with a function. This can be done using pattern matching and a type that contains all possible action types that can be in an edge. The specific cases there are will depend on the language. In ExtWhile, an edge may contain either a skip ($skip$), a boolean expression (production rule b), an assignment ($x := a$) or an array assignment ($A[a_1] := a_2$).

2.1 Syntax of ExtWhile

The following syntactic categories are used to define ExtWhile:

a	$\in$	$AExp$	arithmetic expressions
b	$\in$	$BExp$	boolean expressions
S	$\in$	$Stmt$	statements
x	$\in$	Var	variables
n	$\in$	Num	numerals
A	$\in$	Arr	arrays
op_a	$\in$	Op_a	binary arithmetic operators
op_{au}	$\in$	Op_{au}	unary arithmetic operators
op_b	$\in$	Op_b	binary boolean operators
op_{bu}	$\in$	Op_{bu}	unary boolean operators
op_r	$\in$	Op_r	relational operators

where

$$
\begin{array}{lcl}
Op_a & = & \{\ +,\ -,\ *,\ /\ \} \\
Op_{au} & = & \{\ -\ \} \\
Op_b & = & \{\ \&\&,\ ||\ ,\ \&,\ |\ \} \\
Op_{bu} & = & \{\ !\ \} \\
Op_r & = & \{\ >,\ >=,\ <,\ <=,\ =,\ <>\ \}
\end{array}
$$

The syntax of ExtWhile is given in Definition 2.1:

Definition 2.1: Syntax for ExtWhile statements S using the following BNF notation:

$$
\begin{array}{lcl}
S & ::= & x := a \mid A[a_1] := a_2 \mid \textbf{skip} \mid S_1; S_2 \mid \\
 & & \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od} \\
a & ::= & x \mid n \mid A[a] \mid a_1\ op_a\ a_2 \mid op_{au}\ a \mid (a) \\
b & ::= & \textbf{true} \mid \textbf{false} \mid op_{bu}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid (b)
\end{array}
$$

Variables and arrays must match the regular expression [a-zA-Z][a-zA-Z0-9]*.

2.1.1 Example ExtWhile Program

As a demonstration of the language, the code for getting the xth Fibonacci number is in Listing 2.1. Note that the conditional inside the while can be removed by modifying the loop condition to $x > 1$, however it is left non-optimised to demonstrate

the grammar.

```
x := 3;
y := 1;
if x <= 0 then
    y:= 0
else
    while x > 0 do
        if x = 1 then
            skip
        else
            y := y*x
        fi;
        x := x-1
    od
fi;
A[0] := y
```

Listing 2.1: Calculating Fibonacci for 3

2.2 Program Graphs

Analysis of programs is done using program graphs.

Definition 2.2: A program graph consists of the following:

1. **Q**: A finite set of nodes which are all associated with a unique integer (for example $\{q_0; q_1; q_2\}$)
2. **Act**: A finite set of actions
3. $\mathbf{E} \subseteq \mathbf{Q} \times \mathbf{Q} \times \mathbf{Act}$: A finite list of edges
4. $q_{\triangleright}$, $q_{\blacktriangleleft} \in \mathbf{Q}$: Two nodes called the *initial* node and *final* node

An edge $\{Q_1 = q_\circ;\ Q_2 = q_\bullet;\ Action = \alpha\}$ has source node $q_\circ$, target node $q_\bullet$ and is labelled with action α.

It is required that all nodes are reachable from $q_{\triangleright}$, and that $q_{\blacktriangleleft}$ is reachable from all nodes.

It is also required by the developed framework that the *initial* node is identified by integer 0 and the *final* node is contained in Q_2 in the final node in the edge list **E**.

Construction of program graphs in ExtWhile is specified by the *edges* function, which can be found in Definition 2.3 and Definition 2.4. The function can also be seen in graphical form in Figure 2.1

Definition 2.3: Edge Construction for ExtWhile in simple statements

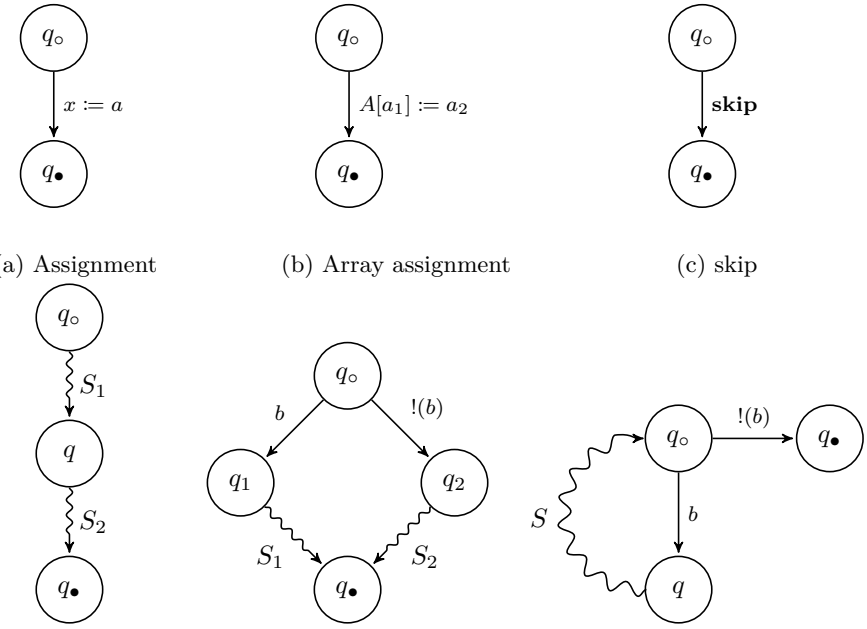
$$\begin{aligned}
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![x \coloneqq a]\!] &= [\{\ Q_1 = q_\circ; \\
&\quad\ Q_2 = q_\bullet; \\
&\quad\ \mathit{Action} = x \coloneqq a\}] \\
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![A[a_1] \coloneqq a_2]\!] &= [\{\ Q_1 = q_\circ; \\
&\quad\ Q_2 = q_\bullet; \\
&\quad\ \mathit{Action} = A[a_1] \coloneqq a_2\}] \\
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![\mathbf{skip}]\!] &= [\{\ Q_1 = q_\circ; \\
&\quad\ Q_2 = q_\bullet; \\
&\quad\ \mathit{Action} = \mathbf{skip}\}] \\
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![S_1; S_2]\!] &= \text{let } q \text{ be fresh} \\
&\quad\ E_1 = \mathbf{edges}(q_\circ \rightsquigarrow q)[\![S_1]\!] \\
&\quad\ E_2 = \mathbf{edges}(q \rightsquigarrow q_\bullet)[\![S_2]\!] \\
&\quad \text{in } E_1 \mathbin{@} E_2
\end{aligned}$$

where a fresh node is a node that has not been used in this program before, and @ represents list concatenation.

Definition 2.4: Edge Construction for ExtWhile in complex statements

$$
\begin{array}{lcl}
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}]\!] & = & \text{let } q_1 \text{ and } q_2 \text{ be fresh} \\
 & & \quad E_1 = \mathbf{edges}(q_1 \rightsquigarrow q_\bullet)[\![S_1]\!] \\
 & & \quad E_2 = \mathbf{edges}(q_2 \rightsquigarrow q_\bullet)[\![S_2]\!] \\
 & & \text{in } [\{\ Q_1 = q_\circ; \\
 & & \qquad Q_2 = q_1; \\
 & & \qquad Action = b\}; \\
 & & \quad \{\ Q_1 = q_\circ; \\
 & & \qquad Q_2 = q_2; \\
 & & \qquad Action = !(b)\}] \\
 & & \quad @\ E_1\ @\ E_2 \\
 & & \\
\mathbf{edges}(q_\circ \rightsquigarrow q_\bullet)[\![\mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}]\!] & = & \text{let } q \text{ be fresh} \\
 & & \quad E = \mathbf{edges}(q \rightsquigarrow q_\circ)[\![S]\!] \\
 & & \text{in } E\ @\ [\{Q_1 = q_\circ; \\
 & & \qquad\qquad Q_2 = q; \\
 & & \qquad\qquad Action = b\}; \\
 & & \qquad \{Q_1 = q_\circ; \\
 & & \qquad\ Q_2 = q_\bullet; \\
 & & \qquad\ Action = !(b)\}]
\end{array}
$$

where a fresh node is a node that has not been used in this program before, and @ represents list concatenation.



(a) Assignment (b) Array assignment (c) skip

(d) Sequencing (e) if else (f) while do

Figure 2.1: Construction of Program Graphs for different statements

2.2.1 Example program graph

As a demonstration of the program graph construction function *edges*, the program graph for the Fibonacci code in Listing 2.1 can be seen in Figure 2.2.

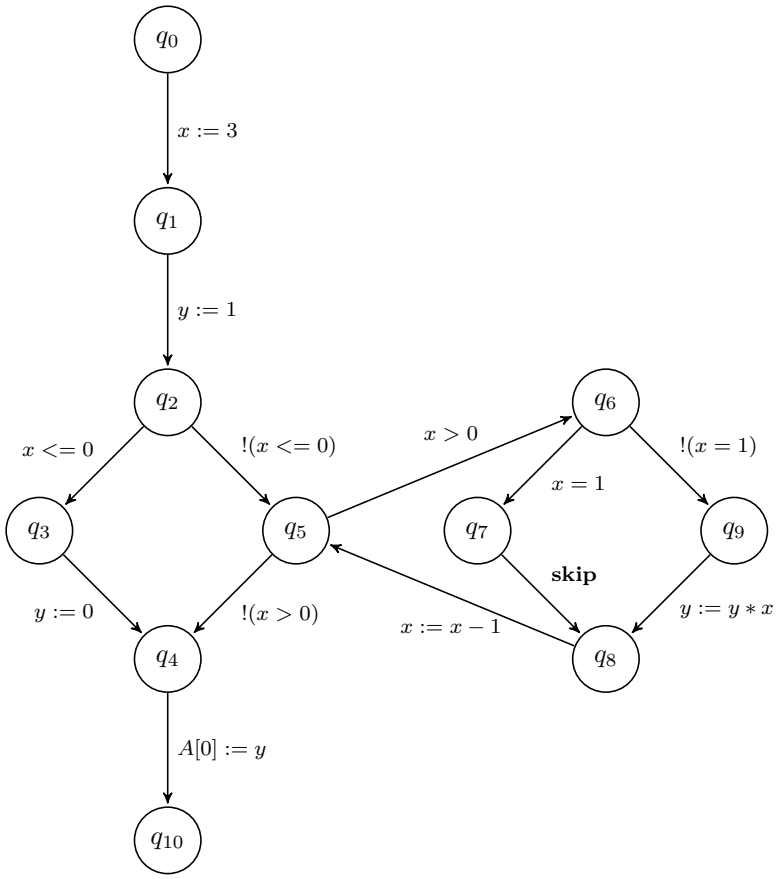


Figure 2.2: Program graph for Fibonacci as in Listing 2.1

CHAPTER 3

Program Analysis

3.1 The Monotone Framework and Program Analysis

The monotone framework is an underlying structure that exists in some analyses, which allows developers to use generic algorithms to solve equations or constraints, and thus reach a stable solution within finite time. This report follows the definitions in [NNH15] and [NN18], while making small modifications to adapt the definitions to the specific case that is being treated in this report.

An analysis in this context is a mapping from each node in a program graph to a property space or domain $\hat{\sigma}$.

$$Analysis \ : \ \mathbf{Q} \rightarrow \hat{\sigma}$$

The property space will describe some properties or states of the program, for example the parity a variable, or which expression results may already have been computed.

While $\hat{\sigma}$ is the domain, $\hat{\sigma}_n$ is written to represent the value of $\hat{\sigma}$ associated with node n (for example $\hat{\sigma}_0$ or $\hat{\sigma}_1$).

Analyses in the monotone framework may be classified with respect to two categories:

- Direction: An analysis may evaluate program graphs using a similar flow to how execution is done (Forward), or reverse the direction, thus effectively reversing the edges (Backward).
- Operation: An analysis may require the greatest sets that solve the constraints (LUB), or the least sets that solve the equations (GLB). These are also known as may and must analyses respectively.

An analysis must have:

- Direction: Forward or Backward, which causes the direction of edges to change if defined as backwards.
- Operation: LUB or GLB, which defines if the analysis should use the least upper bound or greatest lower bound when merging results, when the analysis

stabilises, and the analysis information which all non initial analysis nodes start with.

- S: Initial analysis node, which can be retrieved using the function in (3.1).
- iota (ι): Specification of analysis information at S ($\iota \in \hat{\sigma}$).
- Transfer Function ($\hat{S}$): A transfer function ($\hat{S}[\![e]\!] : \hat{\sigma} \to \hat{\sigma}$).
- Edges: List of edges forming the program graph, where the initial node is $Node\ 0$ and the last node is the inbound node in the last edge of the list.

$$S = \begin{cases} 0 & \text{if } direction = Forward \\ Last(Edges).Q2 & \text{if } direction = Backward \end{cases} \tag{3.1}$$

where *Last* returns the last element of a list.

The result of an analysis is then defined as an equation:

$$\sigma_n = \bigsqcup \{\hat{S}[\![\{m;\ n;\ \alpha\}]\!]\hat{\sigma}_m \mid \{m;\ n;\ \alpha\} \in Edges\} \sqcup \iota_S^n$$

where $\bigsqcup$ may either be the least upper bound or the greatest lower bound, depending on the operation of the analysis. ι_s^n is defined by:

$$\iota_S^n = \begin{cases} \iota & n = S \\ init & otherwise \end{cases}$$

where

$$init = \begin{cases} \bot & \text{if } operation = LUB \\ \top & \text{if } operation = GLB \end{cases}$$

Equivalently, the result of the analysis can be expressed as constraints for all edges:

$$\hat{S}[\![\{a;\ b;\ \alpha\}]\!]\sigma_a \sqsubseteq \sigma_b$$

and

$$\iota \sqsubseteq \sigma_S$$

$$init \sqsubseteq \sigma_n \text{ when } n \neq S$$

where the ordering $\sqsubseteq$ depends on the operation of the analysis. For example in sets it will be $\subseteq$ if it is LUB and $\supseteq$ if its GLB.

3.2 Analysis Domain

In order for the domain to be accepted in the monotone framework, it must be a pointed semi-lattice. For this project, we increase the requirements to demand that the domain is a complete lattice. This is to ensure that domains can be used for both must and may analyses. It is also expected from the domain that it satisfies both

Definition 3.1: Complete Lattice
A lattice is a partially ordered set $(\hat{\sigma}, \sqsubseteq)$ in which all subsets have least upper bounds and greatest lower bounds.

the ascending and descending chain conditions (finite lattice height). This ensures that no matter the analysis direction, all $\hat{\sigma}_n$ will eventually stabilise.

Definition 3.2: Ascending and Descending Chain Condition
A complete lattice satisfies the ascending chain condition if it is impossible to create an infinite increasing chain in the lattice.

$$l_1 \sqsubset l_2 \sqsubset l_3 \sqsubset l_4 ... \tag{3.2}$$

A complete lattice satisfies the descending chain condition if it is impossible to create an infinite decreasing chain in the lattice.

$$l_1 \sqsupset l_2 \sqsupset l_3 \sqsupset l_4 ... \tag{3.3}$$

3.3 Analysis Functions

In order for the transfer functions to be accepted in the monotone framework, there must be a transfer function for every type of edge.

Also each transfer function must be monotone.

Definition 3.3: Monotonicity
A function is monotonic if it does not alter ordering when applied to elements.

$$\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$$

3.4 Analysis specification

In order to run an analysis in the developed framework of this project the user must provide the following inputs:

- An ExtendedWhile program, written in the file `Program.extW`.
- A MetaL domain $\hat{\sigma}$, written in the file `Domain.metaL`.
- Direction: A variable called **direction**, set to either Forward or Backward, written in the file `TransferFunction.fs`.
- Operation: A variable called **operation**, set to either LUB or GLB, written in the file `TransferFunction.fs`.
- Iota: A variable **iota** of the type *sigma*, which was generated by the framework in `Domain.fs`, written in the file `TransferFunction.fs`.
- Transfer functions for each type (**TF_Boolean**, **TF_Assignment**, **TF_Skip** and **TF_ArrayAssignment**) which receive a tuple $(sigma * Edge)$ and return the type *sigma*.
- QuickChecking parameters: Three sets of parameters that will be used to test the monotonicity of the function, written in the file `QCheckParameters.txt`.

Depending on the domain, the user might also need to input:

- Combination and comparison functions for the defined domain set:

$$\textbf{subset\_s} : (type * type) \rightarrow boolean$$

$$\textbf{superset\_s} : (type * type) \rightarrow boolean$$

$$\textbf{union\_s} : (type * type) \rightarrow type$$

$$\textbf{intersect\_s} : (type * type) \rightarrow type$$

- Bottom element: A variable **bot** of type *sigma*, containing the bottom element of the lattice.
- Top element: A variable **top** of type *sigma*, containing the top element of the lattice.

CHAPTER 4

Domain Specification

As discussed in Chapter 3, the monotone framework works inside a domain $\hat{\sigma}$ which is a complete lattice when using $\sqsubseteq$ and $\sqsupseteq$; and where all subsets of the lattice have both a least upper bound $\sqcup$ and a greatest lower bound $\sqcap$.

One of the objectives of the developed framework is to facilitate the fast creation of analyses, by automatically generating data types, comparison operators and combination operators. To achieve this, the project has defined a language in which to describe domains which in some cases can be guaranteed to be complete lattices, and where the framework has generated code so the domain can be used for both must and may analyses, without having to rewrite code.

The defined language, and by extension the framework, works best when using domains which are what later will be defined as *Powerset Comparison Domain*s, but in order to provide flexibility, and reduce the limitations of the framework it also allows domains which are more complex, and thus will change the behaviour of the framework.

4.1 MetaL Syntax

The syntax of MetaL can be seen in Definition 4.1.

Simple sets (**SSet**) are the sets which are code dependant, that is the elements in them depend in the program that is being analysed in the framework. **VAR** is the set of variables. For example Listing 2.1 would result in the following set:

$$VAR = \{Var\ \text{“}x\text{”}; Var\ \text{“}y\text{”}\} \tag{4.1}$$

ARR is the set of all arrays, which in Listing 2.1 is:

$$ARR = \{Arr\ \text{“}A\text{”}\} \tag{4.2}$$

IDENT is the equivalent of a union of variables and arrays, in MetaL represented by:

$$[\ \mathbf{VAR}\ \cup\ \mathbf{ARR}\ ] \tag{4.3}$$

Definition 4.1: Syntax of a domain (*Domain*) in MetaL

$$
\begin{array}{lll}
Domain & ::= & DInstance \mid Domain * DInstance \\
DInstance & ::= & P(Set) \mid [\ SSet \rightarrow DInstance\ ] \mid [Set] \\
SSet & ::= & VAR \mid ARR \mid IDENT \mid Q \\
Set & ::= & SSet \mid Set * Set \mid [\ SSet \rightarrow Set\ ] \mid P(Set) \mid [\ Set \cup Set\ ] \\
 & & \mid \{List\} \mid INT \mid STR \\
List & ::= & element; list \mid element
\end{array}
$$

All *element* objects must be unique in the domain to avoid wrong type inference, and must match the regular expression [A−Z][a−zA−Z0 − 9_]* to satisfy the F# Discriminated Unions format, with the exception of the reserved strings **P**, **Q**, **IDENT**, **VAR**, **ARR**, **INT** and **STR**.

Similarly to a regular union expression in MetaL, identifier results in a Discriminated Union type with case identifiers *Var1* and *Arr1*, which using example sets (4.1) and (4.2) leads to:

$$IDENT = \{Var1\ (Var\ \text{“}x\text{”}); Var1\ (Var\ \text{“}y\text{”}); Arr1\ (Arr\ \text{“}A\text{”})\} \tag{4.4}$$

Q is the set of all nodes occurring in the program, which in Listing 2.1 and Figure 2.2 is:

$$\begin{aligned} Q = \{&Node\ 0; Node\ 1; Node\ 2; Node\ 3; Node\ 4; Node\ 5; \\ &Node\ 6; Node\ 7; Node\ 8; Node\ 9; Node\ 10\} \end{aligned} \tag{4.5}$$

In order to use one of these simple sets in the domain, it is required that they are not equivalent to the empty set. For example the domain to analyse the program in Listing 2.1 would not be allowed to contain **ARR** if the last line of code was omitted (since there would be no arrays in the code), but it may use any other **SSet**. This limitation comes from the usage of the FsCheck tool to generate only valid elements in these sets, and the fact that if there are no elements that can be selected, then the generator cannot produce any valid element. The framework always has data types for these sets, which can be found in `/Framework/Types.fs`, and seen in Listing 4.1. Since the types are already in the code, it is not needed to generate a type. Only unions and records will create instances of **SSet**s. Code generation for **INT** and **STR** is also not necessary.

```
(* Analysis Program Meta Types*)
type Node = Node of int
type Var = Var of string
type Arr = Arr of string
type Ident =
| Var1 of Var
| Arr1 of Arr
```

Listing 4.1: **SSet** types in the framework

The operator $*$ represents the Cartesian product of either two domains (**Domain** $*$ **DInstace**) or two sets (**Set** $*$ **Set**). Thus a domain **Domain** may either be a single domain **DInstance** or a complex domain by having the Cartesian product of several domains. The implementation of this framework allows the Cartesian product of two or three domains, but can be extended finitely. This limitation is a consequence of the definition of lattice operations $\sqcup$ (join operator), $\sqcap$ (meet operator); and partial ordering relations $\sqsubseteq$ and $\sqsupseteq$; which are automatically generated by the framework. The generated type for the Cartesian product of domains is a tuple, as seen in Listing 4.2, while the generated type for the Cartesian product of sets is a record, as seen in Listing 4.3. Note that identifiers are freshly generated (unique labels).

```
type ComplexDomain = Powerset1 * Powerset2
```

Listing 4.2: Example of generated type for the Cartesian product of domains $\mathbf{P}(\_) * \mathbf{P}(\_)$

```
type Record1 = {
    Ident1 : Ident ;
    Union1 : Union1;
}
```

Listing 4.3: Example of generated type for the Cartesian product of sets **Ident** $*$ [$\_ \cup \_$]

A single domain may be a **P**(**Set**), which represents the powerset of the set inside, a total function space [**SSet** $\rightarrow$ **DInstace**] or a set [**Set**]. If the single domain is a powerset, then it may not have an instance of an infinite set happening inside, since this would be a violation of the ascending chain condition. MetaL has two possible ways of making a set infinite: Using the **INT** set which holds an arbitrary integer, or the **STR** set which holds an arbitrary string. The generated code for a powerset will use the F# set type, as can be seen in Listing 4.4. The generated code for total function spaces will use the F# map type, as can be seen in Listing 4.5.

```
type Powerset1 = Node Set
```

Listing 4.4: Example of generated type for a Powerset $\mathbf{P}(\mathbf{Q})$

```
type Map1 = Map<Var, Map2>
```

Listing 4.5: Example of generated type for a TFS [**VAR** $\rightarrow$ [_ $\rightarrow$ _]]

The usage of a [**Set**] domain means that the domain is not comparable using automatically generated functions (since it is not a powerset), thus the framework user will have to provide functions to associate an ordering to the elements, so that it may be a complete lattice. This is discussed further in Section 4.3. Furthermore the usage of a [**Set**] domain disables the usage of complex domains (**Domain** $*$ **DInstace**), since this would require custom operations for each instance of [**Set**], and the framework would have to automatically use them correctly. A future implementation could extend the framework, in the same way that the maximum size of complex domains can be extended. This would however not add any expressiveness of the language, since the Cartesian product of domains can be done inside the single domain, using the Cartesian product of sets rule and total function space, set and powerset rules within the set definition. This can be seen in Definition 4.2. The [**Set**] rule does not produce a data type, since it is only an indicator of a *Graph Domain* starting. A possibility for further development would be to generate an alias for the graph type, so it is explicit, similarly to how the data type *sigma* is created.

Definition 4.2: Transformation from Cartesian domain into a simple domain, when one of the domains is [**Set**].

$$[\mathbf{Set}] * \mathbf{DInstance} = [\mathbf{Set} * Tr(\mathbf{DInstace})]$$

and

$$\mathbf{DInstance} * [\mathbf{Set}] = [Tr(\mathbf{DInstace}) * \mathbf{Set}]$$

where

$$Tr : \mathbf{DInstance} \rightarrow \mathbf{Set}$$

is defined by

$$Tr(d) = \begin{cases} \mathbf{P}(set) & \text{when } d = \mathbf{P}(set) \\ set & \text{when } d = [set] \\ [sset \rightarrow Tr(dN)] & \text{when } d = [sset \rightarrow dN] \end{cases}$$

where Tr will not infinitely recurse because of the finite specification of the domain, and thus finite amount of recursions.

Additionally, sets may be a list of strings representing elements in the set, for example Signs Analysis could use the domain:

$$[\ \mathbf{IDENT} \rightarrow \mathbf{P}(\{Minus; Zero; Plus\})\ ] \tag{4.6}$$

The generated data type is a discriminated union, as can be seen in Listing 4.6.

```
type List1 =
    | A
    | B
    | C
```

Listing 4.6: Example of generated type for a List $\{A; B; C\}$

The final possibility is to use the union ($\cup$) operator on two sets, which would simply append the new elements to the set, as opposed to getting the combinations as in a Cartesian product. For example a Constant Propagation Analysis could use the domain:

$$[\ \mathbf{VAR} \rightarrow [\ [\{Bot; Top\} \cup \mathbf{INT}]\ ]\ ] \tag{4.7}$$

The generated data type is also a discriminated union, as seen in Listing 4.7.

```
type Union1 =
    | Int1 of int
    | List1 of List1
```

Listing 4.7: Example of generated type for a Union $[\ \mathbf{INT} \cup \{\_\}\ ]$

A summary of the constraints on the specification of the domain in MetaL can be seen in Definition 4.3.

Definition 4.3: Invalid domains inside and outside the MetaL syntax

1. Cartesian product of two domains (**Domain** $*$ **DInstance**): This may only happen if both domains do not have any instance of the $DInstance$ rule $[Set]$ in them, since this would require two or more user specified operation functions $(\sqsubseteq, \sqsupseteq, \sqcup, \sqcap)$
2. Powerset (**P**(**Set**)): Powersets may not contain either **INT** or **STR**, since they are in theory infinite sets. If it is desired to have the powerset of a limited amount of numbers, it is possible to create elements for each, and use the list format ({**list**}).
3. Simple Sets (**SSet**): Simple sets may only be used if they are non-empty. The **Q** set will always be non-empty, since the empty program is not allowed in ExtWhile. **VAR**, **ARR** and **IDENT** can be empty sets, for example in a program consisting of only *skip*.

(4.) Total Function Space with an infinite set as key: If a total function space was allowed to have an infinite set as key, then it would break the ascending chain condition. This case is outside the syntax, since total function spaces are only allowed to happen with simple sets, which are finite as a consequence of the analysed program being finite.

All these rules will automatically be verified by the Framework, notifying the user if a rule is broken.

4.2 Isomorphism of domains and analysis variants

When defining analyses, there are often many isomorphic domains that can be used in order to achieve the same results, in slightly different formats. The change in domain might change the underlying data structure implementation of $\hat{\sigma}$, as well as the functions used to compare ($\sqsubseteq$ and $\sqsupseteq$) and combine ($\sqcup$ and $\sqcap$) states. There may also be domains which encapsulate additional information, for example the independent vs. relational analysis domains.

An example of the isomorphic nature of domains is (4.8), in which results can be passed freely without loosing any information. This can be done by using the transformation in Definition 4.4. The symbol $\cong$ is used to represent isomorphism.

$$[set_1 \to \mathbf{P}(set_2)] \cong \mathbf{P}(set_1 * set_2) \tag{4.8}$$

An example of two domains which are not isomorphic because one contains additional information can be seen in (4.9). This is because one of the analysis domains is

Definition 4.4: Transformations between $\mathbf{P}(set_1 * set_2)$ and $[set_1 \rightarrow \mathbf{P}(set_2)]$
Given

$$D_1 \in P(set_1 * set_2)$$

$$D_2 : [set_1 \rightarrow P(set_2)]$$

The transformation from D_1 to D_2 is given by:

$$D_2(s_1) = \{s_2 \mid (s_1, s_2) \in D_1\}[NN18]$$

The transformation from D_2 to D_1 id given by:

$$D_1 = \bigcup_{s_1 \in set_1} \{(s_1, s_2) \mid s_2 \in D_2(s_1)\}$$

where set_1 is finite in METAL because it can only be an instance of an **SSet**.

independent in respect to set_1, while the other contains information about the relation between the different elements in set_1.

$$P([set1 \rightarrow set_2]) \ncong P(set_1 * set_2) \tag{4.9}$$

Other cases of isomorphism in METAL are caused by commutativity of the $*$ operator (Definition 4.5), commutativity and associativity in the $\cup$ operator (Definition 4.6) and commutativity of list sets (Definition 4.7).

Definition 4.5: Transformations between $set_1 * set_2$ and $set_2 * set_1$
Given

$$D_1 : (s_1, s_2), \text{ where } s_1 \in set_1 \wedge s_2 \in set_2$$

$$D_2 : (s_2, s_1), \text{ where } s_1 \in set_1 \wedge s_2 \in set_2$$

To go from one domain to the other:

$$D_1 = (a_1, a_2) \text{ where } D_2 = (a_2, a_1)$$

$$D_2 = (a_2, a_1) \text{ where } D_1 = (a_1, a_2)$$

Definition 4.6: Isomorphism of the $\cup$ operator

Associativity in the $\cup$ operator:

$$[\,[\,set_1\ \cup\ set_2\,]\ \cup\ set_3\,] \cong [\,set_1\ \cup\ [\,set_2 \cup\ set_3\,]\,]$$

Commutativity in the $[\,set_1\ \cup\ set_2\,]$ rule:

$$[\,set_1\ \cup\ set_2\,] \cong [\,set_2\ \cup\ set_1\,]$$

Both these isomorphisms come from the fact that set union is associative and commutative, and the fact that the framework interprets domains using the $\cup$ operators to be the union of both cases.

Definition 4.7: Isomorphism in lists

$$Element1; Element2 \cong Element2; Element1$$

Since a *List* is an enumeration of possible values in a set, the order does not alter the set that is being defined.

Definition 4.2 is also an example of isomorphic domains, where in this case one domain form is not allowed in MetaL. As specified by (4.3), there is also isomorphism between **IDENT** and [**VAR** $\cup$ **ARR**]. This is because **IDENT** is an implementation of [**VAR** $\cup$ **ARR**] that is always generated, in order to allow this in the **SSet** rule, but not allow any arbitrary union in it.

In terms of the generated types, the isomorphism in Definition 4.4 will result in a different type structure (a map instead of a set), and as a consequence the generated comparison and combination operators will be different. The first change will affect how the user will write transfer functions, since they must return the type that has been generated for sigma. The generated operators on the other hand will not affect the user, as it is all done by the framework automatically, unless it is within a custom comparison set (see Section 4.3 for more information on this).

In Definition 4.5 the type structure will be the same, except that the evaluation order will be changed, thus affecting the type names. For example if both sets are lists, then the first element will always be $ListN$ and the next one will be $ListM$ where $M = N + 1$. The operators will, like in the previous case be automatically generated, unless within a custom comparison set.

In Definition 4.6 the type will not change the numbering in associativity, but it

will change which elements are contained in each union type. The resulting types can be seen in Listing 4.8 and Listing 4.9, where *eval* represents the further evaluation of a set.

```
type Union2 =
    | eval(set1)
    | eval(set2)

type Union1 =
    | Union2 of Union2
    | eval(set3)
```

Listing 4.8: Generated type structure for first case

```
type Union2 =
    | eval(set2)
    | eval(set3)

type Union1 =
    | eval(set1)
    | Union2 of Union2
```

Listing 4.9: Generated type structure for second case

For commutativity, the type structure is changed in a similar way to how commutativity in Cartesian products work.

In Definition 4.7 the type will be written in a different order, but be equivalent, since the order of cases in an F# discriminated union does not affect the type.

4.2.1 Domain Specification for Reaching Definitions

As an example of how an analysis can be formulated in many different domains (which are not always isomorphic); five different domains for the reaching definitions analysis will be proposed.

Reaching definitions is an analysis which records the last program point where each variable and array may have been modified. Traditionally this is done by connecting variables and arrays with two nodes, and the possibility of a question mark if the variable/array has not been changed since the program start. The user input for each discussed possible domain is included in the framework source code under the `Examples/Analysis` folder, and the domain folder is in brackets.

4.2.1.1 Reaching Definitions Domain 1 (`Examples/Analysis/RD1/`)

$$\mathbf{P}(\mathbf{IDENT} * [\mathbf{Q} \cup \{\mathbf{QM}\}] * \mathbf{Q}) \tag{4.10}$$

This domain one of the closest possible MetaL domains to the domain suggested in [NN18]. It simply uses the **IDENT SSet** instead of the equivalent union.

4.2.1.2 Reaching Definitions Domain 2 (`Examples/Analysis/RD2/`)

$$\mathbf{P}(\mathbf{VAR} * [\mathbf{Q} \cup \{\mathbf{QM1}\}] * \mathbf{Q}) * \mathbf{P}(\mathbf{ARR} * [\mathbf{Q} \cup \{\mathbf{QM2}\}] * \mathbf{Q}) \tag{4.11}$$

Alternatively, instead of using the union, the cases for variables and arrays could be split into each their own domain, and the Cartesian product could be used. This domain could be used if the user desires to easily separate arrays from variables. It does however require that neither **VAR** or **ARR** are empty sets, while the previous domain just requires one of the sets to be non-empty. Also this shows that even though $QM1$ and $QM2$ express the same idea, they must have different labels, due to the fact that elements must be unique in the domain.

4.2.1.3 Reaching Definitions Domain 3 (`Examples/Analysis/RD3/`)

$$\mathbf{P}(\mathbf{VAR} * [\mathbf{Q} * \mathbf{Q} \cup \{\mathbf{QM1}\}]) * \mathbf{P}(\mathbf{ARR} * [\mathbf{Q} * \mathbf{Q} \cup \{\mathbf{QM2}\}]) \tag{4.12}$$

This domain is technically not isomorphic to the previous domains, but it is still a valid domain for reaching definitions thanks to the fact that $QM1$ and $QM2$ do not require a node, since it represents no change since the start of the program. This domain thus removes invalid possibilities inside the type specification, such as having (Var "x", List1 QM1, Node 4) which evidently is not a valid element in the analysis.

4.2.1.4 Reaching Definitions Domain 4 (`Examples/Analysis/RD4/`)

$$[\mathbf{VAR} \to \mathbf{P}([\mathbf{Q} * \mathbf{Q} \cup \{\mathbf{QM1}\}])] * [\mathbf{ARR} \to \mathbf{P}([\mathbf{Q} * \mathbf{Q} \cup \{\mathbf{QM2}\}])] \tag{4.13}$$

Using the isomorphism specified in Definition 4.4, the domain can be transformed to a map.

4.2.1.5 Reaching Definitions Domain 5 (`Examples/Analysis/RD5/`)

$$[\mathbf{IDENT} \to \mathbf{P}([\mathbf{Q} * \mathbf{Q} \cup \{\mathbf{QM}\}])] \tag{4.14}$$

It is also possible to use the **IDENT SSet** when using maps.

4.3 Domain Types and operators

There are two main types of domains which can be specified in MetaL, and which domain type is used will affect both the required inputs, and the guarantees that the framework provides. The first type is what will be referred as a *Powerset Comparison Domain*. It is a domain in which the [**Set**] **DInstance** rule is not used, thus ensuring that every **DInstance** is either a powerset $\mathbf{P}(\mathbf{Set})$ or a sequence of total function

spaces [**SSet** $\rightarrow$ **DInstance**] ending in a powerset. An example of this could be an alternative Reaching Definitions domain:

$$[\mathbf{IDENT} \rightarrow [\mathbf{Q} \rightarrow \mathbf{P}([\mathbf{Q} \cup \{\mathbf{QM}\}])]] \tag{4.15}$$

the Signs analysis domain: (`Examples/Analysis/Sign1/`)

$$[\mathbf{IDENT} \rightarrow \mathbf{P}(\{Plus; Zero; Minus\})] \tag{4.16}$$

and a simple reachability domain:

$$\mathbf{P}(\{Reachable\}) \tag{4.17}$$

These domains have the advantage that comparison operators ($\sqsubseteq$ and $\sqsupseteq$) and combination operators ($\sqcup$ and $\sqcap$); which are specified in Definition 4.8, can be automatically generated, following the specification in Definition 4.9 for powersets, Definition 4.11 for total function spaces and Definition 4.10 for Cartesian product domains.

Definition 4.8: Specification of comparison and combination operators

The monotone framework requires the use of operations to compare and combine two lattice states. The operation used will depend on the specification of the analysis (if the analysis is a must or a may analysis).

$$\begin{aligned} \sqsubseteq &: type \times type \rightarrow boolean \\ \sqsupseteq &: type \times type \rightarrow boolean \\ \sqcup &: type \times type \rightarrow type \\ \sqcap &: type \times type \rightarrow type \end{aligned}$$

The implementation of these operators will depend on the domain type generated, and might require operations on different layers of the type structure.

Definition 4.9: Automatic generation of operators for powersets

For a powerset, standard set operations can be used. F# code is shown in blue.

$$\sqsubseteq_{pw}\ (p_1, p_2) = p_1 \subseteq p_2 = \textit{Set.isSubset}\ p_1\ p_2$$

$$\sqsupseteq_{pw}\ (p_1, p_2) = p_1 \supseteq p_2 = \textit{Set.isSuperset}\ p_1\ p_2$$

$$\sqcup_{pw}(p_1, p_2) = p_1 \cup p_2 = \textit{Set.union}\ p_1\ p_2$$

$$\sqcap_{pw}(p_1, p_2) = p_1 \cap p_2 = \textit{Set.intersect}\ p_1\ p_2$$

Definition 4.10: Automatic generation of operators for Cartesian product domains

In Cartesian product domains, the generated type structure is a tuple with two or three elements. Operations are done independently on each domain, and then results are merged. For a two element tuple:

$$\sqsubseteq_{p2}\ (\sqsubseteq_{n1}, \sqsubseteq_{n2})\ ((d_{11}, d_{12}), (d_{21}, d_{22})) = (\sqsubseteq_{n1}\ (d_{11}, d_{21}))\ \wedge\ (\sqsubseteq_{n2}\ (d_{12}, d_{22}))$$

$$\sqsupseteq_{p2}\ (\sqsupseteq_{n1}, \sqsupseteq_{n2})\ ((d_{11}, d_{12}), (d_{21}, d_{22})) = (\sqsupseteq_{n1}\ (d_{11}, d_{21}))\ \wedge\ (\sqsupseteq_{n2}\ (d_{12}, d_{22}))$$

$$\sqcup_{p2}\ (\sqcup_{n1}, \sqcup_{n2})\ ((d_{11}, d_{12}), (d_{21}, d_{22})) = (\sqcup_{n1}\ (d_{11}, d_{21}),\ \ \sqcup_{n2}\ \ (d_{12}, d_{22}))$$

$$\sqcap_{p2}\ (\sqcap_{n1}, \sqcap_{n2})\ ((d_{11}, d_{12}), (d_{21}, d_{22})) = (\sqcap_{n1}\ (d_{11}, d_{21}),\ \ \sqcap_{n2}\ \ (d_{12}, d_{22}))$$

Defining the operations for larger cases follows the same structure.

For example in domain (4.11) the generated operation for join would be

$$\sqcup_{\hat{\sigma}} = \sqcup_{p2}\ (\sqcup_{pw}, \sqcup_{pw})\ ((d_{11}, d_{12}), (d_{21}, d_{22}))$$

Definition 4.11: Automatic generation of operators for total function spaces
For total function spaces, the maps must have the same keys, and the operation must hold for the values of each key in the maps:

$$\sqsubseteq_m\ (\sqsubseteq_n)\ (m_1, m_2) = \begin{cases} \bigwedge\limits_{s \in k(m_1)} & \sqsubseteq_n\ (m_1(s), m_2(s)) & \text{if } k_{eq}(m_1, m_2) \\ false & & \text{otherwise} \end{cases}$$

$$\sqsupseteq_m\ (\sqsupseteq_n)\ (m_1, m_2) = \begin{cases} \bigwedge\limits_{s \in k(m_1)} & \sqsupseteq_n\ (m_1(s), m_2(s)) & \text{if } k_{eq}(m_1, m_2) \\ false & & \text{otherwise} \end{cases}$$

$$\sqcup_m\ (\sqcup_n)\ (m_1, m_2) = \begin{cases} res & \text{if } k_{eq}(m_1, m_2) \text{ and} \\ & \forall s \in k(m_1) :\ res(s) = \sqcup_n\ (m_1(s), m_2(s)) \\ error & \text{otherwise} \end{cases}$$

$$\sqcap_m\ (\sqcap_n)\ (m_1, m_2) = \begin{cases} res & \text{if } k_{eq}(m_1, m_2) \text{ and} \\ & \forall s \in k(m_1) :\ res(s) = \sqcap_n\ (m_1(s), m_2(s)) \\ error & \text{otherwise} \end{cases}$$

where k_{eq} returns true if and only if when one of the maps contains a mapping from a key to a value, then the other map must also have a mapping from the same key to a value; and k returns the set of keys used in a map.

From the definition, it can be seen that to compare maps, it is necessary to pass a function as a parameter (op_n) that will be used to compare each map entry individually. For example in the domain in (4.16) would result in the join operator

$$\sqcup_\sigma = \sqcup_m\ (\sqcup_{pw})\ (m_1, m_2)$$

while in the domain in (4.15) the join operator would be

$$\sqcup_\sigma = \sqcup_m\ (\sqcup_m(\sqcup_{pw}))\ (m_1, m_2)$$

since there are two consecutive map structures.

In *Powerset Comparison Domains*, it is also possible to automatically generate lattice states for top and bottom. The framework does this by using the FsCheck generator functionality, and this is discussed in depth in Section 5.5.2.2.

The second domain type will be referred to as *Graph Domains*. These are domains which use the [**Set**] **DInstance** rule and thus may also use the **INT** and **STR** sets. Unlike in Powersets, Maps and Cartesian Domains there is no ordering assumed in a [**Set**] statement, since it is treated as a list of lattice states. For example in the constant propagation domain proposed in (4.7), there is no way for the framework to predict which ordering was meant by the user. Thus the user must specify his own ordering to be associated with the data type generated for the list of states, as well as combination operators.

When a *Graph Domain* is used, the framework offers a reduced capability in some aspects.

While a *Powerset Comparison Domain* will always be a complete lattice (as argued in Section 4.4 and Section 4.5), a *Graph Domain* will only be a complete lattice if the user specified ordering does indeed form a complete lattice. It is possible to expand the framework to test properties such as:

$$\begin{aligned}
\forall a \in L : \bot \sqsubseteq a \wedge a \sqsubseteq \top \quad & (\bot/\top \text{ is lower/upper bound}) \\
\forall a, b \in L : a \sqcup b = b \sqcup a \wedge a \sqcap b = b \sqcap a \quad & (\sqcup, \sqcap \text{ commutative}) \\
\forall a, b, c \in L : (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \quad & (\sqcup \text{ associative}) \\
\forall a, b, c \in L : (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \quad & (\sqcap \text{ associative}) \\
\forall a \in L : a \sqcup a = a \wedge a \sqcap a = a \quad & (\sqcup, \sqcap \text{ idempotent}) \\
\forall a, b \in L : a \sqcup (a \sqcap b) = a \quad & (\sqcup\text{-}\sqcap \text{ absorption}) \\
\forall a, b \in L : a \sqcap (a \sqcup b) = a \quad & (\sqcap\text{-}\sqcup \text{ absorption}) \\
\forall a, b \in L : a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \quad & (\sqsubseteq\text{-}\sqcup\text{-}\sqcap \text{ compatible})
\end{aligned}$$

where L is a lattice with ordering $\sqsubseteq$, a bottom element $\bot$, a top element $\top$ and join ($\sqcup$) and meet ($\sqcap$) operators.

This is proposed and developed by [MM17] for static analyses of the Lua language written in OCaml, and could be a natural extension to the framework.

An additional check which is not discussed is that all subsets must have least upper bounds and greatest lower bounds, as required by the complete lattice definition. This additional constraint is however not as simple to verify, since it requires the framework to distinguish valid values in the lattice data type from invalid states. An example of this can be seen in (4.10) and in (4.11), where element

$(Var1\ (Var\text{“}x\text{”}), List1\ QM, Node\ 5)$ is an element in the lattice, but is not a valid assignment, since a question mark can only be linked to $Node\ 0$. Since this is a *Powerset Comparison Domain*, then it will still be a complete lattice when using set ordering. In a *Graph Domain* it may however cause some issues. For example when doing an Interval analysis, one could use the MetaL domain

$$[\mathbf{IDENT} \rightarrow [[\{Bot\} \cup [\mathbf{INT} \cup \{NegInf\}] * [\mathbf{INT} \cup \{PlusInf\}]]]]$$

and analyse numbers between -20 and 20, and using the infinite values for other numbers. Since QuickChecking works by generating as many possible values as possible in the type structure given, it will generate values outside the range such as for example -30 or 45. It is thus important that transfer functions and custom defined operations ($\sqsubseteq_s$, $\sqsupseteq_s$, $\sqcup_s$, $\sqcap_s$) take into account that they might get valid values in the data type, but invalid states in the lattice. This shows a significant difference in *Graph Domain*s: a lattice is contained by a type, but the type contains more possibilities than just the lattice elements.

In practice this can be dealt with by having the functions interpret invalid lattice elements as bottom ($\bot$), as shown in Section 5.4.2. Failure to cover all cases would lead to a MatchFailureException being thrown, and would then prevent the QuickChecking module from successfully running. This will however skew the distribution of value generation in the QuickChecking module. If a function is incomplete, the F# compiler will warn about incomplete pattern matches. A possible extension to the framework could be allowing the user to specify custom generators. This idea is further discussed in Section 5.5.2.2.

In *Graph Domain*s it is also no longer possible to automatically define a *bottom* ($\bot$) and *top* ($\top$) element just using the data type specification, as was done in *Powerset Comparison Domain*s. It is then up to the user to correctly specify these lattice elements in the Transfer Function Specification. It is possible to verify that these elements are indeed bot and top by using QuickChecking.

The final difference in the framework is that the efficiency of the QuickChecking module will be affected by the use of **INT** and **STR**, as the amount of cases to be tested will increase significantly, and many of the generated values might not be useful to test. For example an alternative MetaL domain to (4.17) could be

$$[STR] \tag{4.18}$$

Where the string is the either the empty string “” or “Reachable” in transfer functions, and the ordering follows Figure 4.1. This would cause the QuickChecking module to generate as many possible strings to verify monotonicity, while the *Powerset Comparison Domain* would know that there are only two possible values in the lattice: the empty set, and the set containing the element *Reachable*.

Figure 4.1: Reachability Lattice: *Graph Domain*

For these reasons, whenever it is possible and viable, it is preferable to use *Powerset Comparison Domain*s, over *Graph Domain*s.

4.3.1 Example: Operator specification in Constant Propagation (`Examples/Analysis/ConstantPropagation/`)

The user will have to specify an ordering as shown in Figure 4.2. The operations generated by the framework are shown in Definition 4.12, the generated type is in Listing 4.10 and the F# code for the operations and top/bottom is in Listing 4.11.

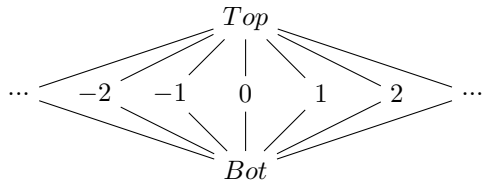


Figure 4.2: Constant Propagation Lattice

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> [ VAR -> [ [{Bot; Top} U INT ] ] ]
*)


type List1 =
    | Bot
    | Top


type Union1 =
    | List1 of List1
    | Int1 of int


```

```
type Map1 = Map<Var,Union1>

type sigma = Map1

type AnalysisResult = Map<Node, sigma>
```

Listing 4.10: Generated type structure for Constant Propagation

Definition 4.12: Generated Operations for Constant Propagation

The framework generates the operations up until the [**Set**] match, resulting in the functions:

$$\begin{aligned} subsetOP\ x &= subset\_m\ (subset\_s)\ x \\ supersetOP\ x &= superset\_m\ (superset\_s)\ x \\ unionOP\ x &= union\_m\ (union\_s)\ x \\ intersectOP\ x &= intesect\_m\ (intersect\_s)\ x \end{aligned}$$

where *subset* represents $\sqsubseteq$, *superset* represents $\sqsupseteq$, *union* represents $\sqcup$ and *intersect* represents $\sqcap$. *subset_m* follows the specification in Definition 4.11.

The user has to define the *subset_s*, *superset_s*, *union_s* and *intersect_s* functions.

```
let bot : sigma = genSigma Variables (List1 Bot)
let top : sigma = genSigma Variables (List1 Top)

let subset_s (set1 : Union1, set2 : Union1) : bool =
    match (set1, set2) with
    | (x, y) when x = y            -> true
    | (List1 Top, _)               -> false
    | (_, List1 Top)               -> true
    | (Int1 x, Int1 y) when x<>y   -> false
    | (List1 Bot, _)               -> true
    | (_, List1 Bot)               -> false
    | _                            -> failwith "Error: Failed to match case"

let superset_s (set1 : Union1, set2 : Union1) : bool =
    match (set1, set2) with
    | (x, y) when x = y            -> true
    | (List1 Bot, _)               -> false
    | (_, List1 Bot)               -> true
    | (Int1 x, Int1 y) when x<>y   -> false
    | (List1 Top, _)               -> true
```

```
21      | (_, List1 Top)                -> false
22      | _                            -> failwith "Error: Failed to match case"
23
24  let union_s (set1 : Union1, set2 : Union1) : Union1 =
25      match (set1, set2) with
26      | (x, y) when x = y            -> set1
27      | (List1 Bot, _)               -> set2
28      | (_, List1 Bot)               -> set1
29      | (Int1 x, Int1 y) when x<>y    -> List1 Top
30      | (List1 Top, _)               -> set1
31      | (_, List1 Top)               -> set2
32      | _                            -> failwith "Error: Failed to match case"
33
34  let intersect_s (set1 : Union1, set2 : Union1) : Union1 =
35      match (set1, set2) with
36      | (x, y) when x = y            -> set1
37      | (List1 Bot, _)               -> set1
38      | (_, List1 Bot)               -> set2
39      | (Int1 x, Int1 y) when x<>y    -> List1 Bot
40      | (List1 Top, _)               -> set2
41      | (_, List1 Top)               -> set1
42      | _                            -> failwith "Error: Failed to match case"
```

Listing 4.11: User specified operations on lattice

While the specification of operators is technically defining the lattice, and thus part of the domain specification, it is unnatural to express it within the MetaL syntax, since it is a very different concept. It is more natural to expect the user to provide F# code as in Listing 4.11, and simply prepend it to the Transfer Function Specification file.

4.4 Complete Lattices

As previously mentioned, it is only guaranteed that domains in MetaL are a complete lattice if they are *Powerset Comparison Domains*. Removing the syntax from only *Graph Domains*, what remains is:

$$
\begin{array}{lcl}
\mathit{Domain} & ::= & \mathit{DInstance} \mid \mathit{Domain} * \mathit{DInstance} \\
\mathit{DInstance} & ::= & P(\mathit{Set}) \mid [\ \mathit{SSet} \rightarrow \mathit{DInstance}\ ] \\
\mathit{SSet} & ::= & \mathit{VAR} \mid \mathit{ARR} \mid \mathit{IDENT} \mid Q \\
\mathit{Set} & ::= & \mathit{SSet} \mid \mathit{Set} * \mathit{Set} \mid [\ \mathit{SSet} \rightarrow \mathit{Set}\ ] \mid P(\mathit{Set}) \mid [\ \mathit{Set} \cup \mathit{Set}\ ] \\
 & & \mid \{\mathit{List}\} \\
\mathit{List} & ::= & \mathit{element}; \mathit{list} \mid \mathit{element}
\end{array}
$$

It is known from [NNH15] that a Powerset of a specific set will always be be a complete lattice when matched with either the $\subseteq$ or $\supseteq$ ordering. Since the framework generates the ordering automatically to be one of these options, then we can state that using the Powerset rule $\mathbf{P}(\mathbf{Set})$ will never lead to a non complete lattice.

It can be seen from the grammar that a **DInstance** will always eventually end with a Powerset, due to the finite specification of the domain. It can be concluded that this will always lead to a complete lattice, due to the fact that it is known that given a complete lattice $L_1 = (L_1, \sqsubseteq_1)$, and a finite set S, we can define:

$$L = \{f : S \to L_1 \mid f \text{ is a total function}\}$$

and

$$f \sqsubseteq f' \text{ iff } \forall s \in S : f(s) \sqsubseteq_1 f'(s)$$

and it is a complete lattice. Since the framework automatically generates the ordering, it is ensured that whenever using the Total Function Space rule [$\mathbf{SSet} \to \mathbf{DInstance}$], the resulting domain will always lead to a complete lattice.

Finally, the cartesian product of two complete lattices $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$:

$$L = \{(l_1, l_2) \mid l_1 \in L_1 \wedge l_2 \in L_2\}$$

is a complete lattice when using ordering

$$(l_{11}, l_{21}) \sqsubseteq (l_{12}, l_{22}) \text{ iff } l_{11} \sqsubseteq_1 l_{12} \wedge l_{21} \sqsubseteq_2 l_{22}$$

Once again the framework generates the ordering in accordance to the definition, and thus it will only produce complete lattices.

4.5 Ascending and Descending chain condition

In order for a domain to not satisfy the Ascending and Descending chain conditions, the lattice must either have a looping structure, or infinite height. Since it has been demonstrated in Section 4.4 that all generated domains are complete lattices, then there may not be any loops. In regards to infinite height, this would only be possible if either **SSet** or **Set** can be infinite.

As previously defined **SSet** represents the sets which are extracted depending on the program. The program is finite, given that it has been given as an input in a file, and thus there can only be a finite amount of variables, arrays and nodes in the program graph.

Similarly, a **Set** will be finite due to the finite specification of the domain. An infinite domain could happen for example if there was an infinite list specification,

but it would not be possible to use this as an input domain.

Thus it can be concluded that all domains within this subset of MetaL will satisfy the Ascending and Descending chain conditions.

4.6 Examples

Following are two examples of MetaL domains, and the code that is generated for the domain type and the operations.

4.6.1 Reaching definitions (/Examples/Analysis/RD1/)

$$\mathbf{P}(\mathbf{IDENT} * [\mathbf{Q} \cup \{QM\}] * \mathbf{Q})$$

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P(IDENT * [Q U {QM}] * Q)
*)


type List1 =
    | QM


type Union1 =
    | Node1 of Node
    | List1 of List1


type Record1 = {
    Ident1 : Ident ;
    Union1 : Union1;
    Node2 : Node ;
}

type Powerset1 = Record1 Set

type sigma = Powerset1

type AnalysisResult = Map<Node, sigma>
```

Listing 4.12: RD1 generated domain

As can be seen, the powerset becomes type *Powerset1* which is an F# set of *Record1* elements. The Cartesian product becomes a record (*Record1*), which contains an identifier *Ident1*, a node *Node2* and an element of type *Union1*. *Union1* is the discriminated union of either a node *Node1* or an element of *List1*, which is a discriminated union with only the element *QM*. The type *sigma* is an alias for the data type of the whole domain, and *AnalysisResult* is the data type containing the results (mapping nodes to *sigma*).

```
[<AutoOpen>]
module Operations
let subsetOP x = subset_pw x
let supersetOP x = superset_pw x
let unionOP x = union_pw x
let intersectOP x = intersect_pw x
```

Listing 4.13: RD1 generated operators

The operations will simply use the standard Powerset operations, since sigma is just a set.

4.6.2 Integer Analysis (/Examples/Analysis/Interval/)

$$[\mathbf{IDENT} \rightarrow [[\{Bot\} \cup [\mathbf{INT} \cup \{NegInf\}] * [\mathbf{INT} \cup \{PlusInf\}]]]]$$

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> [ IDENT -> [ [ {Bot} U [INT U {NegInf}] * [INT U {PlusInf}] ] ] ]
*)


type List1 =
    | Bot


type List2 =
    | NegInf


type Union2 =
    | Int1 of int
    | List2 of List2
```

```
21
22
23 type List3 =
24     | PlusInf
25
26
27 type Union3 =
28     | Int2 of int
29     | List3 of List3
30
31
32 type Record1 = {
33     Union2 : Union2;
34     Union3 : Union3;
35 }
36
37 type Union1 =
38     | List1 of List1
39     | Record1 of Record1
40
41
42 type Map1 = Map<Ident,Union1>
43
44 type sigma = Map1
45
46 type AnalysisResult = Map<Node, sigma>
```

Listing 4.14: Interval generated domain

The generated domain is a total function space from identifiers to a union *Union1*. In the generated domain, it becomes *Map1*. *Union1* can either be *Bot*, which is an element in the single case discriminated union *List1*; or an interval, which is generated into a record *Record1*, due to the Cartesian product. The interval is a lower bound union *Union2*, which is either element *NegInf* in *List2*, or an integer *Int1*; and an upper bound which is either *PlusInf* in *List3*, or an integer *Int2*.

```
1 [<AutoOpen>]
2 module Operations
3 let subsetOP x = subset_m (subset_s) x
4 let supersetOP x = superset_m (superset_s) x
5 let unionOP x = union_m (union_s) x
6 let intersectOP x = intersect_m (intersect_s) x
```

Listing 4.15: Interval generated operators

The domain is a total function space, which contains a set domain. The generated operator will perform the predefined map operations, performing the set operations element-wise. The set operations are defined by the user in the Transfer Function specification.

4.7 Extensions

While MetaL allows the user to express many domains thanks to the usage of *Graph Domain*s, there are several ways to extend and improve the domain expressivity. Following are some domains that could be interesting to consider adding to MetaL.

Bounded Integer Domains: The language could be expanded to include an integer type which is known to be within a range. This might be expressed as $INT[X-Y]$, where X and Y are integers within the 32-bit integer type. This would allow for further optimisation, specifically in regards to the efficiency of the QuickChecking generators. An implementation would then automatically generate QuickChecking generators for all bounded integer types automatically, similarly to how the framework deals with maps.

Other domain types: It could be interesting to consider domains using the smash product of several domains, or domains containing monotone function spaces. Some issues that would have to be considered would be how to ensure the generated data type has the correct behaviour, both in terms of automatic generation and dealing with invalid cases.

CHAPTER 5

Transfer Function and Analysis Specification

Transfer functions are the specification of the analysis functions that are used in an analysis. The transfer functions will define the behaviour of the analysis when evaluating edges. The transfer functions will directly affect the analysis of the result, as well as the computational cost of the analysis.

An example of this may be a signs analysis

$$[\ \mathbf{IDENT} \rightarrow \mathbf{P}(\{Minus; Zero; Plus\})\ ]$$

with transfer function

$$\hat{S}[\![b]\!]\hat{\sigma} = \hat{\sigma}$$

compared to a signs analysis which uses transfer function

$$\hat{S}[\![b]\!]\hat{\sigma} = \bigsqcup\{\hat{\sigma}' \in Basic(\hat{\sigma}) \mid true \in \hat{\mathcal{B}}[\![b]\!]\hat{\sigma}'\}$$

where

$$Basic(\hat{\sigma}) = \{\hat{\sigma}' \mid \hat{\sigma}' \sqsubseteq \hat{\sigma} \wedge \forall x \in VAR : |\hat{\sigma}'(Var1\ x)| = 1 \wedge \forall A \in ARR : |\hat{\sigma}'(Arr1\ A)| \geq 1\}$$

and

$$\hat{\mathcal{B}}[\![b]\!] : \hat{\sigma} \rightarrow P(\{true; false\})$$

evaluates a boolean expression b to return the possible boolean outcomes in an input $\hat{\sigma}$.

The first transfer function will give a larger over-approximation, since it will consider that any $\hat{\sigma}$ that reaches the edge goes through, while the second one will filter some cases, at the cost of having to compute the $Basic$ set, and evaluate the boolean expression for each possibility.

From the literature and tool review in Section 1.2 it was seen that within Abstract

Interpretation-based products, there is a high emphasis in ensuring a low false positive percentage. One way of achieving a smaller over-approximation is by using more complex transfer functions. As transfer functions become more complex, it can be harder to prove that they are monotone.

With this in mind the developed framework integrates automatic QuickChecking of transfer functions, to catch some non-monotonic transfer functions as they are developed.

5.1 Transfer Function syntax

Transfer functions can be very complex functions, that may include several intermediary calls to helper functions. It is therefore natural to allow the user to specify them in code, in this case F#. A transfer function

$$\hat{S}[\![e]\!] : \hat{\sigma} \to \hat{\sigma}$$

changes behaviour depending on the edge type of e, so in the framework, the file `TransferFunctions.fs` must contain functions with the following signatures:

val $TF\_Boolean$	$: inSigma : sigma * edge : Edge \to sigma$
val $TF\_Assignment$	$: inSigma : sigma * edge : Edge \to sigma$
val TF_Skip	$: inSigma : sigma * edge : Edge \to sigma$
val $TF\_ArrayAssignment$	$: inSigma : sigma * edge : Edge \to sigma$

The type *Edge* follows the specification in Section 2.2, and the code in Listing 5.1 where *command* is the AST of an action.

```
type Edge =
    {Q1 : Node;
     Q2 : Node;
     Action : command}
```

Listing 5.1: F# Edge type

The type *sigma* is generated by the domain module, following the given domain specification.

Any necessary helper functions may also be declared in the file.

5.2 Analysis Specification Syntax

The `TransferFunctions.fs` file will also contain the analysis specification variables *direction*, *operation* and *iota* with the following signatures:

val *direction*	:	*AnalysisDirection*
val *operation*	:	*AnalysisOp*
val *iota*	:	*sigma*

where AnalysisDirection can be seen in Listing 5.2, AnalysisOp can be seen in Listing 5.3 and sigma is generated by the domain module, following the given domain specification.

```
type AnalysisDirection =
    | Forward
    | Backward
```

Listing 5.2: F# AnalysisDirection type

```
type AnalysisOp =
    | LUB
    | GLB
```

Listing 5.3: F# AnalysisOp type

5.3 Graph Domain Extension

Recall from Chapter 4, that when working in a *Graph Domain*, the lattice ordering cannot be automatically generated. The user will have to specify in the `TransferFunctions.fs` file functions for ordering, as well as the top and bottom elements. The necessary signatures are:

val *subset_s*	:	$set1 : 'a * set2 : 'a \rightarrow bool$
val *superset_s*	:	$set1 : 'a * set2 : 'a \rightarrow bool$
val *union_s*	:	$set1 : 'a * set2 : 'a \rightarrow 'a$
val *intersection_s*	:	$set1 : 'a * set2 : 'a \rightarrow 'a$
val *bot*	:	*sigma*
val *top*	:	*sigma*

where $'a$ is the type generated when matching the [**Set**] rule, and sigma is generated by the domain module, following the given domain specification.

In order to ease use of the framework, and avoid the user assuming the wrong types in the operators, a future version could generate a type abbreviation or alias, much like how sigma is used currently; to have an explicit signature instead of using abstract types.

5.4 Examples

As a demonstration, the transfer function specification for `Examples/Analysis/RD1/` and `Examples/Analysis/Interval/` will be explained.

5.4.1 Reaching Definitions

```
[<AutoOpen>]
module TransferFunctions

// Helper code
let rec genIotaV (vars, oldIota) =
    match vars with
    | [] -> oldIota
    | var::next -> genIotaV (next, (Set.union oldIota (Set.empty.Add({Ident1
        = Var1 var; Union1 = List1(QM); Node2 = Node(0)}))))


let rec genIotaA (arrs, oldIota) =
    match arrs with
    | [] -> oldIota
    | arr::next -> genIotaA (next, (Set.union oldIota (Set.empty.Add({Ident1
        = Arr1 arr; Union1 = List1(QM); Node2 = Node(0)}))))

(*            Analysis Type            *)
// Direction
let direction : AnalysisDirection = Forward
// Combination operator
let operation : AnalysisOp = LUB
// Iota
let iota : sigma = Set.union (genIotaV (Variables, Set.empty)) (genIotaA (
    Arrays, Set.empty))


// Helper code
let rec remove (inSet, outSet, killCond) =
    match inSet with
    | [] -> outSet
    | x::xs when x.Ident1=killCond -> remove (xs, outSet, killCond)
    | x::xs -> remove (xs, (Set.union outSet (Set.empty.Add(x))), killCond)

let getVar ast =
    match ast with
    |   AssignCommand (a, b)         -> a
```

```
35      |   _                              -> failwith "Cannot extract variable
            from non-variable assignment"
36
37  let getArr ast =
38      match ast with
39      |   ArrAssignCommand(a, b, c)   -> a
40      |   _                                -> failwith "Cannot extract variable
             from non-array assignment"
41
42  (*             TRANSFER FUNCTIONS             *)
43  let TF_Boolean (inSigma : sigma, edge : Edge) : sigma = inSigma
44
45  let TF_Assignment (inSigma : sigma, edge : Edge) : sigma = (Set.union (Set.
        empty.Add({Ident1 = Var1 (getVar edge.Action); Union1 = Node1(edge.Q2);
        Node2 = edge.Q1})) (remove ((Set.toList inSigma), Set.empty, (Var1 (
        getVar edge.Action)))))
46
47  let TF_Skip (inSigma : sigma, edge : Edge) : sigma = inSigma
48
49  // May not kill in arrays
50  let TF_ArrayAssignment (inSigma : sigma, edge : Edge) : sigma = (Set.union (
        Set.empty.Add({Ident1 = Arr1 (getArr edge.Action); Union1 = Node1(edge.
        Q2) ;Node2 = edge.Q1})) inSigma)
```

Listing 5.4: `Examples/Analysis/RD1/TransferFunction.fs`

The domain used in RD1 is a *Powerset Comparison Domain*, so the ordering and top/bot are automatically generated.

Following is a description of the code by blocks:

- Lines 1-2: These lines are mandatory, since the framework will load the file, and it is expected that the module is automatically opened
- Lines 4-14: These are helper functions to generate iota where each variable and array is linked to the connection between $Node$ 0 and the question mark $List1(QM)$.
- Lines 16-22: These lines are mandatory. They are the analysis specification, as described in Section 5.2. As can be seen the user uses his access to lists *Variables* and *Arrays*. Lists *Nodes* and *Identifiers* are also available if needed.
- Lines 25-40: These are helper functions used for extracting variables and arrays from an edge AST, and a function to update sigma for assignments.
- Lines 42-50: These lines are mandatory. They are the transfer function specification, as described in Section 5.1.

5.4.2 Interval Analysis

```
[<AutoOpen>]
module TransferFunctions

// Analysis Range
let minI = -10
let maxI = 10

(*      Graph domain Specification      *)

// Helper functions
...

// bot and top elements
let bot : sigma = genSigma Identifiers (List1 Bot)
let top : sigma = genSigma Identifiers (Record1 {Union2 = List2 NegInf;
    Union3 = List3 PlusInf})

printfn "Top is: %A" top
printfn "Bot is: %A" bot

// Validity Check: If inside range, pass, else bottom
let chk (interval:Union1): Union1 =
    match interval with
    | Record1 {Union2 = Int1 x; Union3 = y} when x<minI     -> List1 Bot
    | Record1 {Union2 = x; Union3 = Int2 y} when y>maxI     -> List1 Bot
    | Record1 {Union2 = Int1 x; Union3 = Int2 y} when x>y   -> List1 Bot
    | a                                                     -> a

// Helper operators: Function for each part of the record
...

(*      Custom Operators for a GRAPH domain         *)

// subset op for interval lattice
let subset_s (set1, set2) : bool =
    match (chk(set1), chk(set2)) with
    ...

// superset op for interval lattice
let superset_s (set1, set2) : bool =
    match (chk(set1), chk(set2)) with
    ...

// union op for interval lattice
let union_s (set1, set2) =
    match (chk(set1), chk(set2)) with
    ...

// intersect op for interval lattice
let intersect_s (set1, set2) =
    match (chk(set1), chk(set2)) with
    ...
```

```
139 (*            Analysis Type            *)
140 // Direction
141 let direction : AnalysisDirection = Forward
142 // Combination operator
143 let operation : AnalysisOp = LUB
144 // Iota
145 let iota : sigma = genSigma Identifiers (Record1 {Union2 = List2 NegInf;
        Union3 = List3 PlusInf})
146
147 // Helper code: Operation evaluation
        ...

345 let rec evalA (a:aexp, s:sigma) : Union1 =
346     match a with
347     | VarExpr(v)            -> chk(s.[Var1 v])
348     | NumExpr(i) when i<minI-> Record1 {Union2 = List2 NegInf; Union3 = Int2
            minI}
349     | NumExpr(i) when i>maxI-> Record1 {Union2 = Int1 maxI; Union3 = List3
            PlusInf}
350     | NumExpr(i)            -> Record1 {Union2 = Int1 i; Union3 = Int2 i}
351     | ArrExpr(aname, aex)   -> chk(s.[Arr1 aname])
352     | SumExpr(a1, a2)       -> evalSum ((evalA (a1,s)), (evalA (a2, s)))
353     | MinExpr(a1, a2)       -> evalMin ((evalA (a1,s)), (evalA (a2, s)))
354     | MultExpr(a1, a2)      -> evalMult ((evalA (a1,s)), (evalA (a2, s)))
355     | DivExpr(a1, a2)       -> evalDiv ((evalA (a1,s)), (evalA (a2, s)))
356     | UMinExpr(a)           -> evalUmin (evalA (a,s))
...

374 (*            TRANSFER FUNCTIONS            *)
375 let TF_Boolean (inSigma : sigma, edge : Edge) : sigma = inSigma
376
377 let TF_Assignment (inSigma : sigma, edge : Edge) : sigma =
378     inSigma.Add(Var1 (getVar edge.Action), evalA (getA edge.Action, inSigma)
            )
379
380 let TF_Skip (inSigma : sigma, edge : Edge) : sigma = inSigma
381
382 // May not kill in arrays
383 let TF_ArrayAssignment (inSigma : sigma, edge : Edge) : sigma =
384     inSigma.Add(Arr1 (getArr edge.Action),
385         union_s (evalA (getA edge.Action, inSigma), inSigma.[Arr1 (getArr
                edge.Action)]))
```

Listing 5.5: `Examples/Analysis/Interval/TransferFunction.fs`

The domain used in Interval is a *Graph Domain*, so the user specifies top, bottom and operators for the set.

Following is a description of the code by blocks:

- Lines 1-2: Mandatory module header, same as in the previous example
- Lines 4-6: These are the interval maxima and minima. These values are just needed for interval analysis, and are only used by the user defined functions.

- Lines 10-22: These lines calculate and assign the complete lattice values for top and bottom. These two variables must be specified only in *Graph Domain*s.
- Lines 24-39: This function is used to validate input sets to functions, ensuring only valid lattice elements are used. If the set element it is called with is an invalid element, it will return bottom. This function is need due to the random generation of values by QuickChecking.
- Lines 32-139: Specification of operators in the set. Note that functions call the *chk* function on the parameters to ensure only valid elements are used.
- Lines 139-146: Analysis specification, as described in Section 5.2.
- Lines 147-356: Evaluation of arithmetic expressions. Values already in *sigma* are validated using the *chk* function, since *sigma* could have been generated randomly.
- Lines 374-385: Transfer function specification for interval analysis.

This version of Interval Analysis reads invalid lattice elements as bottom, as can be seen in function *chk*. This means that there will not be any false positives happening in the monotonicity check due to potential integer overflows, or errors happening due to unexpected elements being given to functions.

An alternative approach could be to have a limited sized integer type in the domain, as proposed in Section 4.7. This would allow the QuickChecking module to only produce valid lattice elements, using automatically generated generators.

Another possibility would be to allow the user to write custom generators. This would also solve the scalability issues with the MetaL domain in Figure 4.1 for the lattice in (4.18). These possible extensions are discussed more in depth in Section 5.5.2.2.

5.5 Monotonicity

As specified in Section 3.3, the monotone framework requires that specified transfer functions are monotone. When creating an analysis this would be verified by making a proof.

The developed framework is created to be part of the initial stages of the design of an analysis, where transfer functions are being redefined several times. As a consequence, the user might not have proven the monotonicity of the transfer functions yet.

One of the goals of this framework is to serve as an early filter to detect and warn the user if any transfer function is not monotone. This will be done using QuickChecking.

5.5.1 QuickChecking

QuickChecking is a methodology for performing property-based testing. There are two main components involved:

- Generators: a way of producing arbitrary testing inputs.
- A property: a statement that will be checked.

The property is then used as a testing oracle, which is given a large quantity of automatically generated random inputs, for which the property is verified. As any oracle, QuickChecking does not provide a proof that the property always holds, but only a guarantee that for the tested values it holds, and that no counter example was found [MM17][CH11].

The idea of using randomly generated programs to test static analysers has been shown to be very effective at finding bugs, even in relatively mature static analysers [Cuo+12].

This framework uses the F# FsCheck tool, which as standard tests properties until 100 passing tests are found. This number can be customised using a mutable inside the library, however there is currently no support for this kind of customisation in the developed framework.

5.5.2 QuickChecking Applied

The framework automatically generates an F# script file that can be run to execute the QuickChecking implementation `/Framework/GenQuickChecker.fsx`. The script creates a set of contexts in which the monotonicity will be verified, creates the generators that will be used in each context, and tests the properties.

Ideally the QuickChecking module would be part of the framework execution, and would be run before the analysis results are computed. This cannot be done in the current version of FsCheck because a mutable variable is used to store generators, which means that the standard generators needed for the monotonicity check are lost when the generator functionality is used to generate top and bot earlier. A future version of FsCheck without mutability has been discussed in the FsCheck repository, however the status of FsCheck 3.0 is unknown.

5.5.2.1 Properties

The tested property in this case is the monotonicity of transfer functions. The code to express this property, and test it can be seen in Listing 5.6.

```
let checkMonotonicity1s (x:sigma, y:sigma, e:Edge) = (subsetOP (x, y))  ==>
    (lazy (subsetOP ((TF_Analysis (x, e)), (TF_Analysis (y, e)))))
let checkMonotonicity2s (x:sigma, y:sigma, e:Edge) = (supersetOP (x, y))
    ==> (lazy (supersetOP ((TF_Analysis (x, e)), (TF_Analysis (y, e)))))

printfn "Checking Monotonicity for Small sized programs"
Check.Quick checkMonotonicity1s
Check.Quick checkMonotonicity2s
```

Listing 5.6: Properties checked

The operator ==> specifies that the written property is a conditional property $< Condition >$ ==> $< Property >$, meaning that if the condition holds then the property is tested. The property also uses the *lazy* F# keyword, to ensure that if the condition doesn't hold, the property is not tested.

In the specified properties, the random values will be the parameters x, y and e. Since the edge, and by extension the action and matching transfer function are random, then a function to match a property verification to a transfer function is created, and can be seen in Listing 5.7

```
(*            TF matching               *)
let TF_Analysis (inSigma : sigma, edge : Edge) : sigma =
    match edge.Action with
    | SkipCommand           -> TF_Skip (inSigma, edge)
    | BoolCommand(_)        -> TF_Boolean (inSigma, edge)
    | AssignCommand(_)      -> TF_Assignment (inSigma, edge)
    | ArrAssignCommand(_)   -> TF_ArrayAssignment (inSigma, edge)
```

Listing 5.7: Matching an edge to its transfer function

Monotonicity is checked for both must and may analysis, since they use different functions. This is done because monotonicity will not only depend on the transfer functions, but also the ordering. Since the user provides ordering functions for *Graph Domains*, it could be possible that a non-complete lattice ordering was specified, which would cause transfer functions to be non-monotonic. This error case will also be caught.

It is important to notice that when testing the monotonicity of transfer functions, the two random *sigma* values must be elements in the same complete lattice. This means that the lattice must be the same when generating random elements, in particular the **SSet** sets must contain the same values. Because of this, QuickChecking is done in three user specified contexts, which will all need their own custom generators.

5.5.2.2 Generators

As previously mentioned, monotonicity checks must test *sigma* values that are in the same lattice. This means that **SSet** lattice elements may only generate valid elements in the current context. To ensure this three contexts are defined in which monotonicity will be tested.

Sample sets: Context generation

A context consists of a specification of what variables, arrays, nodes and identifiers exist currently in the analysis.

The user will specify the dimensions of the **SSet** lists in the following the format (`/Framework/QCheckParameters.txt`):

$$(n_1, v_1, a_1)\ (n_2, v_2, a_2)\ (n_3, v_3, a_3)$$

where n are the number of nodes, v are the amount of variables, and a are the amount of arrays, and all values are positive integers.

The framework will take these integers and generate nodes, variables and arrays automatically. Identifier generation will be inferred from generation of variables and arrays.

An example of a generator in context S, which is the first set of values can be seen in Listing 5.8. The *Gen.elements* function is a generator that randomly picks an element from a list, in this case the list of possible nodes/variables/arrays in the context.

```
    static member Node() =
      {new Arbitrary<Node>() with
          override x.Generator = Gen.elements NodesS
          override x.Shrinker t = Seq.empty }
    static member Var() =
      {new Arbitrary<Var>() with
          override x.Generator = Gen.elements VariablesS
          override x.Shrinker t = Seq.empty }
     static member Arr() =
      { new Arbitrary<Arr>() with
            override x.Generator = Gen.elements ArraysS
            override x.Shrinker t = Seq.empty }
```

Listing 5.8: Generators for context S

Generating maps

In the framework, total function spaces will be represented as maps. An issue with

this is that not all maps are total function spaces. For a map to be a total function space, it must have an assignment for every possible key.

$$m : map < SSet,' a > \text{ is a TFS iff } \forall k \in SSet : \exists m(k)$$

```
1    static member Map1() =
2      {new Arbitrary<Map1>() with
3          override x.Generator = createAnalysis IdentifiersS.Length
               IdentifiersS ( Arb.generate<Union1> |> Gen.listOfLength
               IdentifiersS.Length |> Gen.sample 1000000 1).[0]
4          override x.Shrinker t = Seq.empty }
```

Listing 5.9: Generator for a total function space *Map1* from Identifier to Union1 in Small context

From the example map generator in Listing 5.9 it can be seen that in order to automatically generate a generator which produces only valid total function spaces, the program must know:

- Data type names: For example *Map1* in the example.
- The key **SSet** type associated to the type: For example the map in the example maps an *Identifier* to a *Union1*, hence ***IdentifierS*** in the example.
- The context in which it is generating: For example the small context is represented by the letter ***S***, as seen in *Identifier**S***.
- The type that is being mapped to: For example *Union1* in the example, since *Map1* has the type *map<Identifier, Union1>*

The *createAnalysis* function creates a mapping from each element to a generated *Union1* value.

Generating top and bot

Section 4.3 mentioned that bot and top can be automatically generated for *Powerset Comparison Domain*s. To do this, generators for **SSet**s are set as previously defined, to the analysed program **SSet** values. Maps are also assigned custom automatically generated generators as previously defined. Finally, domains must end with a set data type. Thus generators for the set that contains all possible values, and the empty set are used. The set generator for bot can be seen in Listing 5.10

```
1    static member Set() =
2      {new Arbitrary<Set<'a>>() with
3          override x.Generator = Gen.elements [Set.empty]
4          override x.Shrinker t = Seq.empty }
```

Listing 5.10: Generating bot: Sets

Generator efficiency in graphs

Section 5.4.2 specifies ordering and transfer functions where invalid domain elements are converted to bottom. While this makes the monotonicity verification work, it alters the frequency in which lattice elements are generated. In this case the vast majority of generated elements will be the bottom element, causing the QuickChecking module to do more work than needed. This is a consequence of the domain data type containing a large amount of invalid lattice elements. The following two sections will discuss possible approaches to prevent this.

Domain specific generators

Section 4.7 proposed introducing a bounded integer domain type to MetaL. From the definition of the domain the framework would know the minimum and maximum values, and a generator that only produces valid elements could be automatically generated. The F# code that would be generated can be seen in Listing 5.11.

```
1    static member Int1() =
2      {new Arbitrary<Int1>() with
3          override x.Generator = Gen.elements [ min1 .. max1 ]
4          override x.Shrinker t = Seq.empty }
```

Listing 5.11: Custom generator for a bounded int

User defined generators

In cases such as the domain in (4.17) it could be proposed that the user could write a custom generator to improve the QuickChecking performance. A generator for that case can be seen in Listing 5.12.

```
1    static member Str1() =
2      {new Arbitrary<Str1>() with
3          override x.Generator = Gen.elements [ ""; "Reachable"]
4          override x.Shrinker t = Seq.empty }
```

Listing 5.12: User defined generator for reachability

Advanced generator possibilities

A more efficient QuickChecking module would involve generating pairs of *sigma* elements in the same context, instead of independently generating two sigmas. This would mean that each check has its own random context, and thus the user would not need to specify the three contexts.

Ideally the two values generated would already fulfil the precondition. There are however some large technical challenges to do this, mainly that the pair generation will depend on the ordering. This means that instead of generating many cases which do not comply with the precondition, the generator would often lead to a stack overflow if the *Gen.filter* method was used, and it would lead to many *None* results if using the *Gen.tryFilter* method.

5.5.2.3 Results

There are three possible outcomes from the QuickChecking module:

- *Ok, passed 100 tests*: 100 tests where found where the precondition was true, and so was the property.
- *Arguments exhausted after* $\boldsymbol{x}$ *tests*: Only $\mathbf{x}$ cases where found where the precondition held, and the property held in all of them.
- *Falsifiable, after* $\boldsymbol{x}$ *tests*: A case where the precondition held, but the property did not hold was found after $\mathbf{x}$ cases

Since the test cases are random, the result can also vary from run to run. In general, larger domains also lead to fewer cases found where the precondition holds.

As an example to demonstrate that the QuickChecking module can detect that functions are not monotone, the framework is run on RD1, and the *skip* transfer function is set to Listing 5.13, which is similar to the *flip* function in [MM17].

```
let TF_Skip (inSigma : sigma, edge : Edge) : sigma =
    match Set.count inSigma with
    | 0     -> Set.empty.Add({Ident1 = Var1 (Var "x"); Union1 = Node1 (Node
        0); Node2 = Node 1})
    | _             -> Set.empty
```

Listing 5.13: Non monotone transfer function

Running the generated `GenQuickChecker.fsx` file then gives the following output:

```
Checking Monotonicity for Small sized programs
Falsifiable, after 6 tests (0 shrinks) (StdGen (1901932966,296458904)):
Original:
(set [], set [{Ident1 = Var1 (Var "var0");
               Union1 = Node1 (Node 19);
               Node2 = Node 14;}], {Q1 = Node 10;
                                    Q2 = Node 2;
                                    Action = SkipCommand;}
```

CHAPTER 6

Framework Implementation

6.1 Framework Overview

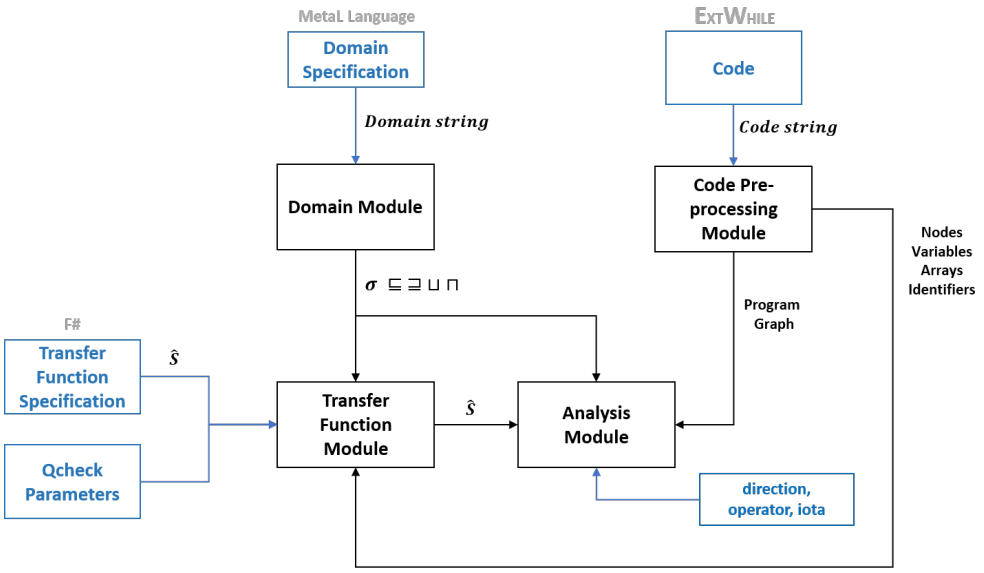


Figure 6.1: Framework module and input overview

A simplified version of the developed framework, and it's modules can be found in Figure 6.1. It shows the inputs specified in Section 3.4 and into which module they are input in the framework. The diagram covers the case where the domain is a *Powerset Comparison Domain*.

Figure 6.2 shows a more detailed overview of the functionality of the framework in terms of functionality instead of modules. In the implementation, functionality is

split in the following way:

- Language dependant blocks (orange): Done by the Code Pre-processing Module, which is described in Section 6.2.
- *Domain Specification* branch, excluding *Graph specification* and its *QuickChecking* functionality: Done by the Domain Module, described in Section 6.3.
- *Transfer Function Specification* branch, *Analysis Specification* branch and *Graph Specification* branch: Done by the Transfer Function Module, described in module Section 6.4.
- *Analysis* and *Worklist Algorithm*: Done by the Analysis Module, described in Section 6.5

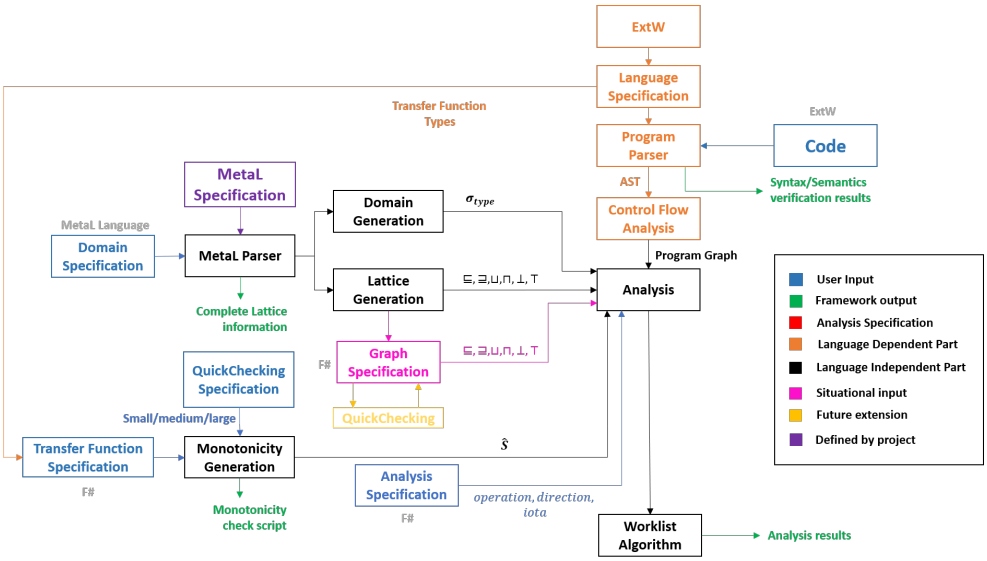


Figure 6.2: Functionality overview of developed framework

6.2 Code Pre-processing Module Implementation

The Code Pre-processing Module is the part of the developed framework which is in charge of all the functionality related to the specific program that is being analysed, and the language it is written in.

In this framework, the language used is ExtWhile, which is described in Chapter 2. The grammar has been transformed in to LexYacc format in `/Framework/`

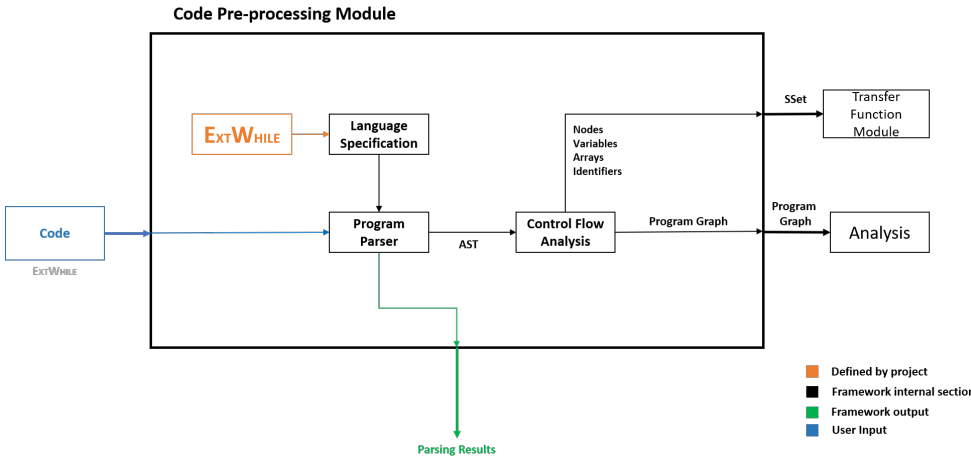


Figure 6.3: Code Pre-processing Module implementation

`ExtWLexer.fsl` for the lexer, and `/Framework/ExtWParser.fsp`. Using the FsLexYacc NuGet package, the specification has been transformed into the generated code `/Framework/ExtWLexer.fs` and `/Framework/ExtWParser.fs`. In order to use these files, the FsLexYacc dll file is used (`/Framework/FsLexYacc.Runtime.dll`). This dll may be reacquired from the original package if needed.

The Code Pre-processing module uses some shared types from `/Framework/Types.fs` file. Once the code is loaded from the `/Framework/Program.extw` file, it is parsed by `/Framework/Parser.fs`, using the *ParseString* function. Due to the fact that FsLexYacc is used, error reporting is not very detailed. On failure to parse the input code, the framework will terminate with message "Error parsing program: Invalid code input". The code is parsed into a list of statements, which contain an AST of the command in the statement and a command type to indicate the context it was found in.

From the statement list, `/Framework/Grapher.fs` transforms it into a list of edges which represent the program graph, using the *GraphStatements* function. This is done following the specification in Definition 2.3 and Definition 2.4.

In the process of parsing, the lists for **SSet**s are acquired, and the results are set to variables *Nodes*, *Variables*, *Arrays* and *Identifiers*. These variables are available to the rest of the framework, and may be used in the domain generation, lattice state calculation and in transfer functions.

Possible future work in this module would be extending the ExtWhile language

and improving error reporting in parsing, in particular returning the code line which could not be matched in the input code.

6.3 Domain Module Implementation

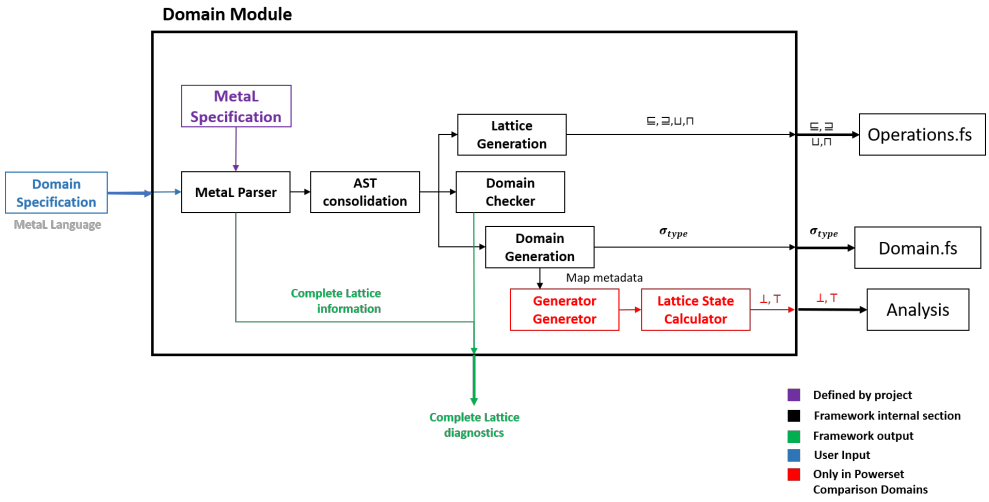


Figure 6.4: Domain Module implementation

The Domain Module is the part of the framework which is in charge of all the functionality related to parsing the MetaL domain, generating a data type specification and operators for it, extracting information about the domain that is needed for QuickChecking, verifying the validity of the Domain and generating and running code to automatically get lattice elements top and bot for *Powerset Comparison Domains*.

The data types for domain ASTs are found in `/Framework/DomainTypes.fs`. The MetaL domain string is loaded from `/Framework/Domain.metaL`, and parsed by `/Framework/DomainParser.fs` using the *ParseStringDom* function, which returns the *domain* type. Parsing is done using the FsLexYacc generated files `/Framework/MetaLLexer.fs` and `/Framework/MetaLParser.fs`, and the FsLexYacc dll (`/Framework/FsLexYacc.Runtime.dll`). To generate them the lexer (`/Framework/MetaLLexer.fsl`) and parser (`/Framework/MetaLParser.fsp`) specifications where used, which are based on the MetaL specification in Definition 4.1.

The parsed domain is then transformed into a more compressed representation. Compression is done in three cases:

1. Cartesian product of domains ($Domain * DInstance$): Several products on the same level are stored as a list of domains, instead of a recursive data type. For example:

   $CartesianDom\ (CartesianDom\ (d1, d2), d3) \Rightarrow CartesianListDom\ [d1; d2; d3]$

   where $d1$, $d2$, $d3$ are not $CartesianDom$ domains.

   This allows the domain type generator to construct the Cartesian product domain tuple ($ComplexDomain$) in a single type, instead dividing it into successive tuples of length two. The uncompressed version can be seen in Listing 6.1, while the compressed version is in Listing 6.2.

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P(Q) * P(Q) * P(Q)
*)


type Powerset3 = Node Set

type Powerset2 = Node Set

type Powerset1 = Node Set

type ComplexDomain2 = Powerset1 * Powerset2

type ComplexDomain1 = ComplexDomain2 * Powerset3

type sigma = ComplexDomain

type AnalysisResult = Map<Node, sigma>
```

Listing 6.1: Uncompressed Cartesian product domain

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P(Q) * P(Q) * P(Q)
*)


type Powerset3 = Node Set

type Powerset2 = Node Set

```

```
type Powerset1 = Node Set

type ComplexDomain = Powerset1 * Powerset2 * Powerset3

type sigma = ComplexDomain

type AnalysisResult = Map<Node, sigma>
```

Listing 6.2: Compressed Cartesian product domain

This compression is a trade-off between user friendly data types, and the ability to automatically generate comparison and combination operators for arbitrary Cartesian product domain lengths.

2. Cartesian product of sets ($Set * Set$): Several products on the same level are stored as a list of sets, instead of a recursive data type. For example:

$$CartesianSet\ (CartesianSet\ (s1, s2), s3) \Rightarrow CartesianListSet\ [s1; s2; s3]$$

where $s1$, $s2$, $s3$ are not $CartesianSet$ sets.

Unlike the Cartesian product of domains, Cartesian products of sets generate an F# record type. The uncompressed generated type is in Listing 6.3, and the compressed generated type is in Listing 6.4.

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P(Q*Q*Q)
*)

type Record2 = {
    Node1 : Node ;
    Node2 : Node ;
}

type Record1 = {
    Record2 : Record2;
    Node3 : Node ;
}

type Powerset1 = Record1 Set

type sigma = Powerset1

type AnalysisResult = Map<Node, sigma>
```

Listing 6.3: Uncompressed Cartesian product set

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P(Q*Q*Q)
*)


type Record1 = {
    Node1 : Node ;
    Node2 : Node ;
    Node3 : Node ;
}

type Powerset1 = Record1 Set

type sigma = Powerset1

type AnalysisResult = Map<Node, sigma>
```

Listing 6.4: Compressed Cartesian product set

3. List elements ($element; List$): Several elements in the same list are compressed into a list, instead of the recursive data type. For example:

$$ListSet\ (LargerList\ (s1, LargerList\ (s2, s3))) \Rightarrow ElemList\ [s1; s2; s3]$$

where $s1$, $s2$, $s3$ are element strings.

The uncompressed generated type can be seen in Listing 6.5, and the compressed generated type can be seen in Listing 6.6.

```
[<AutoOpen>]
module Domain

// Generated Code Section: Domain type
(*
Q -> P({S1;S2;S3})
*)

type List2 =
    | S2
    | S3

type List1 =
    | S1
    | List2


type Powerset1 = List1 Set
```

```
19
20 type sigma = Powerset1
21
22 type AnalysisResult = Map<Node, sigma>
```

Listing 6.5: Compressed list set

```
1
2  [<AutoOpen>]
3  module Domain
4
5  // Generated Code Section: Domain type
6  (*
7  Q -> P({S1;S2;S3})
8  *)
9
10
11 type List1 =
12     | S1
13     | S2
14     | S3
15
16
17 type Powerset1 = List1 Set
18
19 type sigma = Powerset1
20
21 type AnalysisResult = Map<Node, sigma>
```

Listing 6.6: Compressed list set

Using element $S2$ in the compressed generated type would be $List1$ ($List2$ $S2$), while in the compressed type it would be $List1$ $S2$.

Domain compression is generally done to generate data types which are easier to use, and have less usage overhead.

The compressed domain, is used for three tasks. The first task is checking if the domain breaks the first two rules in Definition 4.3. This task is done by `/Framework/DomainChecker.fs`, and will terminate the program if the domain is invalid. The termination error will specify which rule is broken. "Error: Invalid domain (using powerset domain with an infinite set inside)" will be thrown if an **INT** or **STR** is found inside a Powerset (rule 2), and "Error: Invalid domain (Cartesian domain product with unsafe domains)" will be thrown for a violation of rule 1. Rule 3 is checked while the type is being generated by `/Framework/DomainGenerator.fs`, and will terminate with message "Error: Using VAR in a program without any variables", "Error: Using ARR in a program without any arrays", or "Using IDENT in a program without any variables or arrays". The *domCheck* function is used, which returns true if the domain is a *Powerset comparison domain*, and false if it is a *Graph Domain*.

The second task is generating the code for the data type of the domain and is done by `/Framework/DomainGenerator.fs`. The domains will always contain a data type *sigma* which contains the specified MetaL domain, and an *AnalysisResult* type which is a map from program nodes to *sigma*. This type is returned by the analysis module, which is described in Section 6.5. Generation of data types is done following the specification in Section 4.1. The generated domain type is written to `/Framework/Domain.fs`, which is then loaded into the program. The *evaluateAST* function is used, which returns a string with the code of the generated type and information about the maps used in the domain, which are used for automatically generating QuickChecking generators.

The third task is generating ordering and combination functions. `/Framework/LattOps.fs` contains the generic functions for operations, as specified in Definition 4.9, Definition 4.10 and Definition 4.11. `/Framework/CallGenerator.fs` uses the compressed domain to generate functions according to the domain structure. The result is written to `/Framework/Operations.fs`. This file is not immediately loaded, since it may use the *op_s* functions which are defined in the transfer function specification. If the operation file was loaded immediately it might lead to an error due to undefined constructors for *subset_s*, *superset_s*, *union_s* and *intersect_s*.

The first task returned information about the domain type. If the domain was a *Powerset Comparison Domain*, then bot and top have to be automatically calculated by the framework. `/Framework/GeneratorGenerator.fs` has the *outputCode* function, which returns F# code to generate bot and top using the FsCheck package. The code needed to do this task is described in Section 5.5.2.2. The result string is written to `LatticeStates.fs`, which is loaded in, thus calculating variables *top* and *bot*, which will be used by the Analysis Module.

Possible further work in this module would be extending MetaL to allow more domain types, as described by Section 4.7; compressing domains when there are several unions at the same level and reusing previous domain types when patterns are found.

6.4 Transfer Function Module Implementation

The Transfer Function Module is in charge of loading in the user inputs for the Analysis Specification, transfer functions and if in a *Graph Domain*, ordering and combination functions and lattice elements top and bot.

The user inputs are loaded from file `/Framework/TransferFunctions.fs`. The transfer function module will use the *assemble* function in `/Framework/QCheckGenerator.fs` to create a separate script `/Framework/GenQuickChecker.fsx` which can be run to verify the monotonicity of the transfer functions using QuickChecking. This process will use the user provided parameters in `/Framework/QCheckParameters.txt`,

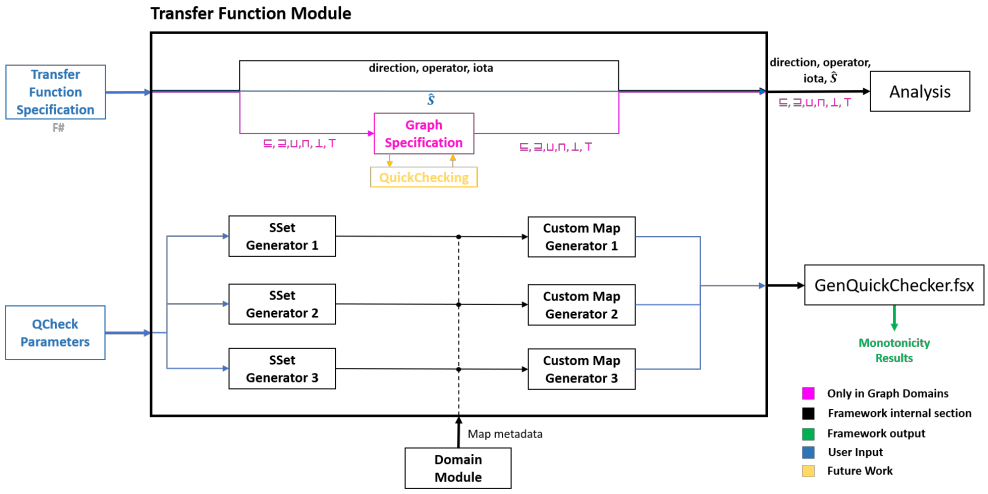


Figure 6.5: Transfer Function Module implementation

as described in Section 5.5.2.2.

Running the generated QuickChecking file will output the results for the verification in the three defined contexts. An example output for the testing in one context could be:

```
Checking Monotonicity for Small sized programs
Arguments exhausted after 36 tests.
Arguments exhausted after 51 tests.
```

Possible future work in this module would be improving the performance of the generated QuickChecking script. Section 5.5.2.2 describes possible changes in the framework generators to reduce the amount of invalid tests done.

6.5 Analysis Module Implementation

The Analysis Module uses the domain specification, transfer function specification, analysis specification; and computes the solution of the static analysis.

The whole analysis functionality is found in `/Framework/Analysis.fs`, and is used by calling the *AnalyseEdges* function with the program graph as parameter. The function will among other things invert edges if the analysis has backward direction, and select the appropriate functions for operation and initial values according to the

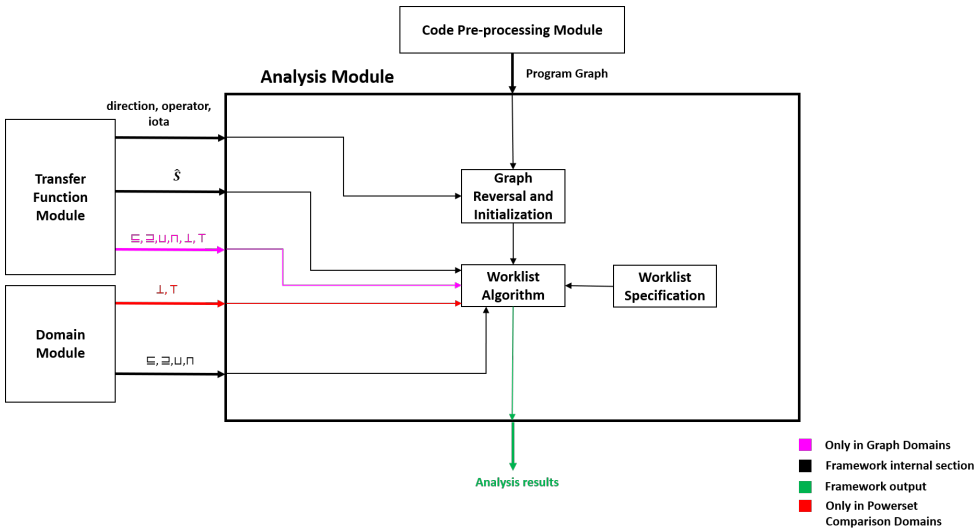


Figure 6.6: Analysis Module implementation

analysis type.

Possible future work in this module would be improving the performance of the analysis by using a more efficient worklist specification, for example reverse post order. The analysis module could also be made more language independent by doing the matching from edges to transfer functions in the transfer function specification, instead of in the analysis module. In practice this would require the user to include the code in Listing 5.7 in the transfer function specification, and the analysis module would simply use the *TF_Analysis* function. It is however not completely natural to have the matching as a user input, due to the fact that it should not be customised. Ideally this task should be done by the language specification.

CHAPTER 7

Conclusion

Recall that this thesis had the objective of creating a design tool to specify analyses within the monotone framework. The purpose of this tool was to simplify the process of creating new analyses, as well as comparing alternative formulations of the same analysis, and making static analyses using Abstract Interpretation more accessible.

To achieve this, the tool generates code automatically, and verifies properties which are required to hold in the monotone framework. Code generation is done for the domain F# type, lattice operations, script generation and generating more precise QuickChecking generators depending on the domain. The framework currently tests the monotonicity of transfer functions by the use of a generated script.

In Chapter 1 an overview of existing solutions is provided, and there is an overview of the functionality the tool provides.

Chapter 2 defines an example language to be analysed (ExtWhile), and discusses how using a different language would affect the framework. The transformation from code to program graphs is a vital part of static analysis within the monotone framework, so this transformation is formalised for ExtWhile.

Chapter 3 describes the monotone framework, applied to the context of this tool. In particular the necessary user inputs are identified and the files used by the framework to receive each input are defined.

Chapter 4 defines how a domain can be specified in the tool, using the syntax given for the MetaL language. The transformation from MetaL domain to F# code is defined and shown in practice. Invalid domains within the language are identified, and rules are made to distinguish correct domains. Isomorphism of domains, is discussed and demonstrated in practice for MetaL. A complete lattice also requires an ordering relation, so the automatic generation of ordering and combination operators is defined. Complex domain cases are also discussed, in particular the challenges that are associated with them, and how the framework deals with these challenges. The chapter also discusses what guarantees are given by the framework in terms of the generated code being a complete lattice, and it satisfying the ascending and descending chain conditions. Finally possible extensions of MetaL are briefly discussed.

Chapter 5 covers the specification of the transfer functions and the analysis parameters, as well as the QuickChecking functionality inbuilt in the tool, both for checking monotonicity and to generate bottom and top elements. The section also defines how to specify operations for *Graph Domain*s.

Chapter 6 is the documentation of the developed F# implementation of the tool. The functionality defined in Chapter 1 is split into modules, and tasks are split into F# files. The key types and functions used by each module are identified. An in depth description of how domains are compressed to create more user friendly data types is provided.

Practical instructions for running the framework can be found in Appendix A.

In conclusion this thesis has created an F# design tool framework as an extension of monotone framework, and defined a language to write domains which are automatically generated into F# data types and functions. Automatic generation of complete lattice operations and lattice elements bottom and top is done in domains which have been identified as *Powerset Comparison Domain*s. These domains are identified using a classification of domains defined by the project, in regard to properties of the generated code. The developed tool automatically generates a script to check the monotonicity of the users transfer functions, using domain dependant automatically generated QuickChecking generators.

Possible future work and extensions for each module were discussed. The framework can be extended in many directions, but one of the most beneficial modifications would be improving the efficiency of the monotonicity QuickChecking by improving the generators.

APPENDIX A

Framework Usage Instructions

The developed framework has been written in F#. The user specifies the inputs in:

- `/Framework/Program.extw`
- `/Framework/Domain.metaL`
- `/Framework/QCheckParameters.txt`
- `/Framework/TransferFunctions.fs`

The framework can be run by executing the `/Framework/FrameworkTest.fsx` F# script file. Executing is done using the F# Interactive program. The code requires the *.dll* files from the FsCheck and FsLexYacc NuGet packages in the same directory. These are already provided in the code, but can be reacquired from the official sources if needed.

To customise the language analysed by the framework (ExtWhile), or the MetaL domain language, the lexers/parsers can be regenerated using the *fslex* and *fsyacc* executable files provided by the FsLexYacc NuGet package. These files are not included in the repository, but can be acquired from the official package source.

To run the monotonicity check, the `/Framework/GenQuickChecker.fsx` script can be run. Note that this file is generated by `/Framework/FrameworkTest.fsx`, thus it will require running the other script first to update it to a new domain.

The repository is preconfigured to run the RD1 analysis (Domain in (4.10)) on the Fibonacci code example (Listing 2.1), using parameters "(20, 8, 4) (200, 100, 50) (2000, 500, 200)", and the monotonicity checking script is pre-generated.

Several examples of analyses are provided in `/Examples/Analysis/`, together with a brief description file. To run an example analysis, simply copy the `Domain.metaL` and `TransferFunctions.fs` files into the `/Framework/` folder.

Bibliography

[And] Paul Anderson. *Detecting Domain-specific Coding Errors with Static Analysis*.

[Bra+14] Guillaume Brat et al. "IKOS: A framework for static analysis based on abstract interpretation". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2014, pages 271–277.

[CC95] Patrick Cousot and Radhia Cousot. "Formal language, grammar and set-constraint-based program analysis by abstract interpretation". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. ACM. 1995, pages 170–181.

[CH11] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Acm sigplan notices* 46.4 (2011), pages 53–64.

[Cuo+12] Pascal Cuoq et al. "Testing static analyzers with randomly generated programs". In: *NASA Formal Methods Symposium*. Springer. 2012, pages 120–125.

[FG13] Christian Ferdinand and AbsInt GmbH. *Abstract Interpretation-based Static Analysis Tools - Proving the Absence of Runtime Errors and Safe Upper Bounds on the Worst-Case Execution Time and Stack Usage*. Conference presentation. 2013. URL: `http://projects.laas.fr/IFSE/FMF/J2/pdf/P04_CFerdinand.pdf`.

[GRA14] Makarand Gawade, K Ravikanth, and Sanjeev Aggarwal. "Constantine: configurable static analysis tool in Eclipse". In: *Software: Practice and Experience* 44.5 (2014), pages 537–563.

[MM17] Jan Midtgaard and Anders Møller. "Quickchecking static analysis properties". In: *Software Testing, Verification and Reliability* 27.6 (2017).

[NN18] Flemming Nielson and Hanne R Nielson. *Program Analysis: An appetizer (Draft)*. 2018.

[NNH15] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[Sch18] Philipp Schubert. *Phasar*. `https://github.com/secure-software-engineering/phasar`. 2018.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ