Data Flow Analysis using Program Graphs

Michal Terepeta

Supervised by
Professor Hanne Riis Nielson
Professor Andrzej Napieralski

Kongens Lyngby 2010
Abstract

Static code analysis is a powerful formal method of analysing software without actually executing any code. It is very commonly used in compilers, but also for bug detection and software verification purposes. Moreover, because of the complexity of today’s software, it is becoming more and more popular as a way for ensuring higher code quality.

Data Flow Analysis is a technique for statically gathering/predicting the possible sets of values at various points in the program, that would arise at the program execution. In the standard approach the analysed program is transformed into a flow graph, where labeled blocks are nodes and edges represent control flow. Such a graph is then the basis for various analyses. This thesis explores the idea of using program graphs for that purpose. In this case the blocks/actions are associated with the edges and the nodes are simply labels. One of the main advantages of this approach is the ability to easily compose multiple graphs and then analyse the resulting one. So by redefining some classical analyses to work on program graphs, it is possible not only to analyse sequential programs, but also shared memory environments with concurrently executing processes.

The thesis defines the program graphs, shows how they can be then merged and presents their semantics. It uses a simple imperative programming language WHILE and elaborates on how programs can be transformed into a program graphs. Moreover it redefines a number of classical analyses to work on such graphs, including Reaching Definitions, Live Variables and Constant Propagation. Finally, some interesting examples of analysing well known algorithms are
The thesis also includes an implementation of the presented ideas and analyses. It is a web application written in Haskell, which helps to demonstrate how the approach works in practice and what are its advantages.
Contents

Abstract

1 Introduction
   1.1 Flow Graphs and Program Graphs
   1.2 Possible advantages

2 Program Graphs
   2.1 Definition
   2.2 Composition
   2.3 Semantics

3 WHILE language
   3.1 Syntax
   3.2 Semantics
   3.3 WHILE programs to Program Graphs
   3.4 Semantic result

4 Program Graphs and Monotone Framework
4.1 Definitions ......................................................... 23
4.2 Worklist algorithm .............................................. 25
4.3 Succinct Solver ..................................................... 26

5 Reaching Definitions and Live Variables .................. 29
  5.1 Reaching Definitions ........................................... 29
  5.2 Live Variables ................................................... 37

6 Constant Propagation .............................................. 41
  6.1 Independent attribute Constant Propagation .............. 41
  6.2 Relational Constant Propagation .............................. 45
  6.3 Comparison ....................................................... 58

7 Interesting examples ............................................ 61
  7.1 Dekker’s algorithm ............................................. 61
  7.2 Peterson’s algorithm ........................................... 64

8 Implementation .................................................... 71
  8.1 Introduction .................................................. 71
  8.2 Using the application .......................................... 72
  8.3 Project structure ............................................... 75

9 Conclusions ....................................................... 79
  9.1 Summary ...................................................... 79
  9.2 Future work ................................................... 80

A Howto install and run the application ...................... 83
  A.1 Requirements .................................................. 83
  A.2 Installing and running ....................................... 84
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The semantics of arithmetic expressions.</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>The semantics of boolean expressions.</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>The semantics of the WHILE language.</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>ALFP syntax.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>ASCII representation of ALFP.</td>
<td>28</td>
</tr>
<tr>
<td>6.1</td>
<td>Arithmetic expressions for CP.</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>Boolean expressions for CP.</td>
<td>43</td>
</tr>
<tr>
<td>6.3</td>
<td>Transfer functions for CP using flow graphs.</td>
<td>44</td>
</tr>
<tr>
<td>6.4</td>
<td>Transfer functions for CP using program graphs.</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Dekker’s algorithm</td>
<td>62</td>
</tr>
<tr>
<td>7.2</td>
<td>Peterson’s algorithm</td>
<td>64</td>
</tr>
<tr>
<td>7.3</td>
<td>Modified Peterson’s algorithm.</td>
<td>65</td>
</tr>
<tr>
<td>7.4</td>
<td>Interleaving where both processes enter the critical section</td>
<td>66</td>
</tr>
<tr>
<td>7.5</td>
<td>Peterson’s algorithm with atomic statements.</td>
<td>66</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Simple example of a flow graph.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Simple example of a program graph</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Graphs for the two simple example programs.</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Composition of the two programs graphs.</td>
<td>4</td>
</tr>
<tr>
<td>5.1</td>
<td>Flow graph for the example program.</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Results of the RD analysis.</td>
<td>33</td>
</tr>
<tr>
<td>5.3</td>
<td>Results of the RD analysis using the Succinct Solver.</td>
<td>36</td>
</tr>
<tr>
<td>5.4</td>
<td>Results of the LV analysis.</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Results of the independent attribute CP analysis.</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Results of the relational CP with $\overline{K} = [x \mapsto 3, \ y \mapsto 3]$.</td>
<td>60</td>
</tr>
<tr>
<td>7.1</td>
<td>Dekker’s algorithm analysed by the independent attribute CP.</td>
<td>63</td>
</tr>
<tr>
<td>7.2</td>
<td>Peterson’s algorithm analysed by the independent attribute CP.</td>
<td>67</td>
</tr>
<tr>
<td>7.3</td>
<td>Peterson’s algorithm analysed by the relational CP.</td>
<td>68</td>
</tr>
<tr>
<td>7.4</td>
<td>Modified Peterson’s algorithm analysed by the relational CP.</td>
<td>69</td>
</tr>
</tbody>
</table>
7.5 Peterson’s algorithm with atomic statements analysed by the relational CP. ........................................ 70

8.1 The main interface of the implementation. ........................................ 73
1.1 Flow Graphs and Program Graphs

In the standard approach to the Data Flow Analysis, such as presented in [3], one uses so called flow graphs for representing the program, that is to be analysed. Such a graph is constructed in the following way:

- each elementary block of a program, such as an assignment or a test, is assigned a unique label and represents a node in the graph
- directed edges are created as to represent all the possible ways the control can flow between the above blocks

As an example consider the following program in the While language:

\[ x := 1; \ (if \ x > 0 \ then \ y := 1 \ else \ y := 0) ; \ z := y \]

The corresponding flow graph is presented in the Figure 1.1. There are a few
things worth noting about flow graphs. First of all, each block can be identified by the corresponding label. Moreover the control flow of the program can be easily represented by the set of pairs of connected blocks. Finally, if one thinks about the graph from the analysis point of view, it is clear that it should be concerned about the results at the entry and exit of each node. In other words, about the state of the analysis before and after a given block is executed. All of these considerations are mentioned here to contrast this approach with the one presented below, namely program graphs.

Program graph is quite similar to a transition system and in this case can be considered simply as an equivalent representation of a program. For similar definition of program graphs and transition systems, but from a model checking perspective, compare [1]. The simple idea behind it is to associate the actions not with nodes but with the edges of the graph. Note that actions are very close to the building blocks used in the flow graphs, we will use the term action when talking about program graphs and building block in case of flow graphs. As the actions are “on” the edges, the nodes are represented only by the labels. And so an edge (with an action) between two nodes represents execution of this action when going from the source to the destination node. The Figure 1.2 presents a program graph for the same example program considered above. This representation has some profound consequences. First of all, the actions are associated with the edges of the graph and thus cannot be identified by a single label - two of them are necessary: one corresponding to the source and one for the destination node. Secondly the graph has two actions for each condition in a program - one corresponding to the true branch and one for false.
1.2 Possible advantages

So there will be more actions in a program graph than the building blocks in an equivalent flow graph. Finally the results of an analysis are associated only with labels (i.e. nodes), not their entry/exit points (from a different perspective one might say that the entry/exit points of an action are explicit - they are the source and destination labels of the edge).

1.2 Possible advantages

Apart from the above, simple observations, the new approach does have quite a few advantages that might not be immediately apparent. One of the most important ones is the ability to easily compose program graphs and thus express concurrently executing processes. Consider for instance the following two programs:

\[\text{Program1} : x := 1\]
Their graphs are extremely simple (Figure 1.3). Now if we consider that the

Figure 1.3: Graphs for the two simple example programs.

programs execute concurrently, we have a starting node that corresponds to the

Figure 1.4: Composition of the two programs graphs.

One of the important things to note about the composition of program graphs is

that they do exhibit the state explosion problem. If in the current example two-
statement programs have been used (e.g. \( x := 1; \ y := 1 \)) then the composed graph would need to have 9 nodes

\[
(1,1), \ (1,2), \ (1,3), \ (2,1), \ (2,2), \ (2,3), \ (3,1), \ (3,2), \ (3,3)
\]

As an advantage, the resulting graph is still just a program graph. Therefore, if we have analyses that can work on such graphs, then they can be used without any changes to analyse systems of concurrently executing programs. Finally we can compose any number of programs and thus analyse systems of any number of processes.
Chapter 2

Program Graphs

2.1 Definition

The previous chapter presented some introduction and rationale behind exploring the idea of using program graphs for Data Flow Analysis. Now before defining them formally, let us first discuss what should be included in such a graph.

First of all, we do want the program graphs to be independent of the programming languages. The main advantage of the program graphs being so generic is that it would allow us to define various analyses only for program graphs and then use them to analyse programs written in different languages. All that would be needed is to define how to transform a program written in some given language into a program graph. Therefore, a program graph should include a set of possible actions, and some semantic function that would provide us with their meaning. Apart from that, we obviously also need some information about the control flow (i.e. some information about edges and the associated actions). Moreover, it is also important to have some initial label that would
indicate where the program starts executing, as well as a final one that would correspond to the end of program execution. Finally, a program graph should be finite - it should have only a finite number of labels and edges.

**Definition 2.1** We define a program graph by the following tuple:

\[(\text{Lab}, \text{Act}, \rightarrow, \mathcal{T}, \text{Lab}_0, \text{Lab}_*)\]

where

- \text{Lab} is a finite set of labels
- \text{Act} is a finite set of actions
- \rightarrow \subseteq \text{Lab} \times \text{Act} \times \text{Lab} is a finite set of arrows
- \mathcal{T} : \text{Act} \rightarrow \text{State} \hookrightarrow \text{State} is a partial semantic function
- \text{Lab}_0 \in \text{Lab} is an initial label
- \text{Lab}_* \in \text{Lab} is a final label

As already mentioned, the actions should correspond to elementary blocks of a program, such as assignments, tests from the conditionals and loops, skip statements, etc. However, we do not define them here - they should be defined along with \(\mathcal{T}\) when introducing a way to convert some programming language into a program graph. Apart from that, note that the sets \text{Lab} and \text{Act} should be easily derivable from the set \(\rightarrow\). It contains all the information about the edges of the graph and the associated actions, which is basically equivalent to the sets of edges and blocks as used in the original flow graph approach. The semantic \(\mathcal{T}\) function captures the semantics of the actions. Note that it is a partial function. Consider for example the following situation: \(\mathcal{T}[\text{false}]\sigma\) - the action might be a condition which is simply false and corresponds to a branch that will not be taken by the program. Thus the result of a function would be probably undefined.

### 2.2 Composition

As already mentioned in the previous chapter, one of the advantages of using program graphs is that it is quite easy to create a graph simulating the execution of multiple concurrent processes. So let us define here the composition operator \(\parallel\) for program graphs.
Definition 2.2 Two program graphs

\[ PG_1 = (\text{Lab}_1, \text{Act}_1, \rightarrow_1, \mathcal{T}_1, \text{Lab}_{o1}, \text{Lab}_*) \]

\[ PG_2 = (\text{Lab}_2, \text{Act}_2, \rightarrow_2, \mathcal{T}_2, \text{Lab}_{o2}, \text{Lab}_*) \]

can be composed into a new graph

\[ PG = PG_1 || PG_2 = (\text{Lab}, \text{Act}, \rightarrow, \mathcal{T}, \text{Lab}_2, \text{Lab}_*) \]

by taking

\[
\begin{align*}
\text{Lab} & = \text{Lab}_1 \times \text{Lab}_2 \\
\text{Act} & = \text{Act}_1 \cup \text{Act}_2 \\
\rightarrow & = \{(\ell_1, \ell_2) \xrightarrow{\text{act}} (\ell'_1, \ell_2) \mid \ell_1 \xrightarrow{\text{act}} \ell'_1 \in \rightarrow_1, \ \ell_2 \in \text{Lab}_2\} \\
& \cup \{(\ell_1, \ell_2) \xrightarrow{\text{act}} (\ell_1, \ell'_2) \mid \ell_2 \xrightarrow{\text{act}} \ell'_2 \in \rightarrow_2, \ \ell_1 \in \text{Lab}_1\} \\
\mathcal{T} & = \mathcal{T}_1 \cup \mathcal{T}_2 \\
\text{Lab}_o & = (\text{Lab}_{o1}, \text{Lab}_{o2}) \\
\text{Lab}_* & = (\text{Lab}_*1, \text{Lab}_*2)
\end{align*}
\]

The new set of labels is simply a Cartesian product of the sets of labels from the separate graphs. The set of actions can be obtained by taking a union of the two initial ones. The most important is how to combine the sets \(\rightarrow\) of the two graphs. In the current, quite simple scenario, it is achieved simply by creating all possible interleavings. The rest of the definition is straightforward.

The most important thing to emphasize here is that the resulting graph does model the concurrent execution of a number of programs. This is so called interleaving paradigm that basically abstracts away from the fact that the programs execute at the same time and considers them from the point of view of the global system state. In such a scenario the concurrency is expressed by the nondeterministic choice of the actions available at the nodes. And so we can create analyses that can work on program graphs and then use them to analyse a system with shared memory concurrency.

Apart from that, note that the above definition does not take into account any form of inter-process communication or synchronisation, as it only creates the interleavings. In a more advanced situation, that would allow it, the process of creating the new \(\rightarrow\) set would have to be extended. Of course this definition allows composition of arbitrary number of processes - the result of the composition of two processes is also a program graph that can be composed with some other one, and so on. However, in practice the resulting program graph can
get quite huge. The number of nodes grows exponentially with the number of programs - consider a program that has \( N \) nodes, then the composition of two such programs will result in a graph with \( N^2 \) nodes, composition of three such programs \( N^3 \), so with \( k \) programs the number of nodes would be \( N^k \).

### 2.3 Semantics

As program graphs are actually only a little bit more complex transition systems, their semantics can be defined in a very straightforward way.

**Definition 2.3** Consider a program graph \((\text{Lab}, \text{Act}, \to, T, \text{Lab}_0, \text{Lab}_*)\). We can define its semantics in the following way

\[
\langle \ell, \sigma \rangle \xrightarrow{\text{act}} \langle \ell', T[act] \sigma \rangle
\]

provided that \( T[act] \sigma \) is defined,

\[
\ell \xrightarrow{\text{act}} \ell' \in \to \quad \text{and} \quad \sigma \in \text{State}
\]

Now it is important to note that the program graphs are a generic representation of programs. Clearly it is possible to express programs written in different programming languages as a program graph. The only requirement is to provide the definition of \( T \) and a way to create the remaining sets (such as \( \to \), \( \text{Lab} \), etc.) along with the initial and final labels for a given program.
3.1 Syntax

In order to demonstrate the analysis techniques using program graphs, we will use a simple imperative programming language called WHILE. Compared to the version introduced in [3], it will be extended to accommodate some additional features, such as the ability to perform a number of assignments atomically.
The abstract syntax is quite simple:

\[\begin{align*}
\text{op}_a & ::= + | - | * | / \\
\text{op}_b & ::= \text{and} | \text{or} \\
\text{op}_r & ::= == | != | > | < \\
a & ::= x | n | a_1 \text{ op}_a a_2 \\
b & ::= \text{true} | \text{false} | \text{not} b | b_1 \text{ op}_b b_2 | a_1 \text{ op}_r a_2 \\
S & ::= x := a | \text{skip} | S_1 ; S_2 | \text{if} b \text{ then } S_1 \text{ else } S_2 | \text{while} b \text{ do } S \\
\text{atomic} & ::= (x := a; ...) \\
P & ::= S_1 || \cdots || S_k
\end{align*}\]

Where \( a \) and \( b \) are arithmetic and boolean expression respectively, \( S \) is a statement and \( P \) represents a **WHILE** program that executes a number of statements concurrently. Please note that the **atomic** statement allows only assignments to be performed within it. This is to avoid problems associated with e.g. having while loops inside it.

Finally it should be noted that the textual representation of the abstract syntax, that will be used throughout the thesis, can be often ambiguous. Consider the following

\[
\text{if } b \text{ then } S_1 \text{ else } S_2 ; S_3
\]

should it be parsed as

\[
\text{if } b \text{ then } \begin{array}{l} S_1 \text{ else } S_2 ; \end{array} \quad S_3
\]

first statement second one

or as

\[
\text{if } b \text{ then } S_1 \text{ else } \begin{array}{l} S_2 ; S_3 \end{array}
\]

false branch

Therefore we will assume that the **if** conditionals and **while** loops will always use a single statement in their branches/body. So the above example would be an if statement followed by \( S_3 \) statement. And if we need to have more than a single statement, we will use parentheses to indicate that, so

\[
\text{while } b \text{ do } (S_1 ; S_2) ; S_3
\]

would mean that \( S_1 \) and \( S_2 \) are executed in the body of the loop and the \( S_3 \) after the loop terminates. Similarly we can use the parentheses in the **if** conditional

\[
\text{if } b \text{ then } (S_1 ; S_2) \text{ else } (S_3 ; S_4) ; S_5
\]

This is somewhat similar to the rule used in the **C** programming language (with **C** using the braces \{ \} instead of parentheses for that purpose).
3.2 Semantics

Having the abstract syntax we can go on and define the formal semantics of the WHILE language. In order to do so we need to first define the semantics of arithmetic (Table 3.1) and boolean (Table 3.2) expressions. We assume that we also have $N$, which defines the semantics of numerals. Moreover $\text{op}_a$, $\text{op}_b$ and $\text{op}_r$ are the semantic counterparts of arithmetic, boolean and relational operators defined by the syntax.

$$A[x] \sigma = \sigma(x)$$
$$A[n] \sigma = N[n]$$
$$A[a_1 \text{ op}_a a_2] \sigma = A[a_1] \sigma \text{ op}_a A[a_2] \sigma$$

Table 3.1: The semantics of arithmetic expressions.

$$B[\text{true}] \sigma = \text{true}$$
$$B[\text{false}] \sigma = \text{false}$$
$$B[\text{not } b] \sigma = \neg B[b] \sigma$$
$$B[b_1 \text{ op}_b b_2] \sigma = B[b_1] \sigma \text{ op}_b B[b_2] \sigma$$
$$B[a_1 \text{ op}_r a_2] \sigma = A[a_1] \sigma \text{ op}_r A[a_2] \sigma$$

Table 3.2: The semantics of boolean expressions.

The semantics of the language itself is presented in the Table 3.3. There are two things that might appear unnecessary at this stage: the done state of a program and the addition of actions on the arrows that express a single step of execution. However, they were added here on purpose and will be quite useful in the next section.
Table 3.3: The semantics of the WHILE language.

### 3.3 While programs to Program Graphs

Having abstract syntax and semantics of our programming language, we should now define how it can be transformed into a program graph. We need to define the semantic function $\mathcal{T}$ and what are the possible actions. Apart from that it is important to specify how to create the set $\rightarrow$ from the abstract syntax that will exactly represent how the program would execute. Note that, while generating this set, we will have to create the necessary labels as we go and associate the right actions with the pairs of labels (i.e. edges). Finally, as already mentioned, the sets $\mathsf{Act}$ and $\mathsf{Lab}$ can be extracted from the $\rightarrow$. 
So let us start with defining the set of possible actions that can be associated with edges in our program graph.

\[ \text{Act} \subseteq \{b, \text{skip}, x := a, \text{read} x, \text{atomic}(x := a; \ldots)\} \]

The \( a \) and \( b \) are consistent with the syntax of WHILE and represent arithmetic and boolean expressions respectively.

Finally we want to define a function that takes a program written in the WHILE language and creates the set \( \rightarrow \). However, as we have to generate the labels in the process and assign correct labels when handling conditions, it needs an additional argument - pair of initial and final labels.

**Definition 3.1** The type of the function \( \text{arrows} \) is

\[ \text{arrows} : \text{Stmt} \rightarrow \text{Lab} \times \text{Lab} \rightarrow \mathcal{P}(\text{Lab} \times \text{Act} \times \text{Lab}) \]

And its definition is given by

\[
\begin{align*}
\text{arrows}[x := a](\ell, \ell') &= \{ \ell \xrightarrow{x := a} \ell' \} \\
\text{arrows}[\text{read} x](\ell, \ell') &= \{ \ell \xrightarrow{\text{read} x} \ell' \} \\
\text{arrows}[\text{atomic}(x := a, \cdots)](\ell, \ell') &= \{ \ell \xrightarrow{\text{atomic}(x := a; \cdots)} \ell' \} \\
\text{arrows}[\text{skip}](\ell, \ell') &= \{ \ell \xrightarrow{\text{skip}} \ell' \} \\
\text{arrows}[S_1; S_2](\ell, \ell') &= \\
&= \text{arrows}[S_1](\ell, \ell_1) \cup \text{arrows}[S_2](\ell_1, \ell') \\
&\text{where } \ell_1 \text{ is fresh} \\
\text{arrows}[\text{while } b \text{ do } S](\ell, \ell') &= \\
&= \{ \ell \xrightarrow{\ell \rightarrow \ell_1, \ell \rightarrow \ell_2} \ell' \} \cup \text{arrows}[S](\ell_1, \ell) \\
&\text{where } \ell_1 \text{ is fresh} \\
\text{arrows}[\text{if } b \text{ then } S_1 \text{ else } S_2](\ell, \ell') &= \\
&= \{ \ell \xrightarrow{\ell \rightarrow \ell_1, \ell \rightarrow \ell_2} \ell' \} \cup \text{arrows}[S_1](\ell_1, \ell') \cup \text{arrows}[S_2](\ell_2, \ell') \\
&\text{where } \ell_1, \ell_2 \text{ are fresh}
\end{align*}
\]

When the function is applied initially to the program, \( \ell \) and \( \ell' \) should be two different labels that will represent the initial and final labels of the whole graph.

Now let us recall that program graphs are defined by a tuple

\[ (\text{Lab}, \text{Act}, \rightarrow, \mathcal{T}, \text{Lab}_0, \text{Lab}_*) \]
With the above we can generate the set $\rightarrow$ (from which \textbf{Lab} and \textbf{Act} can be derived) and the initial/final labels. What is still missing is the semantic function.

**Definition 3.2** We define the semantic function $T$ in the following way

\[
T[\text{skip}]\sigma = \sigma \\
T[x := a]\sigma = \sigma[x \mapsto A[a]\sigma] \\
T[\text{read } x]\sigma = \sigma[x \mapsto z] \\
T[\text{atomic}(ass_1; \ldots; ass_k)]\sigma = (T[ass_k] \circ \cdots \circ T[ass_1])\sigma \\
T[b]\sigma = \begin{cases} 
\sigma & \text{if } B[b]\sigma = \text{true} \\
\text{undef} & \text{otherwise}
\end{cases}
\]

Compare that this is almost the same as the semantics of the \textbf{WHILE} language itself (with the exception of the conditions).

### 3.4 Semantic result

Having defined all of the above, we are in a problematic situation - we have two different semantics: one of the program graphs (as defined in 2.3 using the $T$ function defined in 3.2) and one of the \textbf{WHILE} language itself (Definition 3.3). Because we transform a \textbf{WHILE} program into a program graph, by doing so we switch from using one of the semantics to the other one. Therefore it is comforting to prove that whatever the \textbf{WHILE} program does, the program graph representation will do the same.

Before we go on, we need to define a relation $\approx \rightarrow$ that would express the fact that a currently executing statement in \textbf{WHILE} language and a node (i.e. label) in program graph are related to each other.

**Definition 3.3** The relation $\approx \rightarrow$ is defined as follows

\[
\langle S \rangle \approx \rightarrow \langle \ell \rangle \quad \text{if } \exists \ell' : \text{arrows}[S](\ell, \ell') \subseteq \rightarrow \\
\langle \text{done} \rangle \approx \rightarrow \langle \ell \rangle \quad \text{for all } \ell \\
\langle S_1 \| S_2 \rangle \approx \rightarrow \langle \ell_1, \ell_2 \rangle \quad \text{if } \langle S_1 \rangle \approx_{\rightarrow_1} \langle \ell_1 \rangle \land \langle S_2 \rangle \approx_{\rightarrow_2} \langle \ell_2 \rangle
\]
3.4 Semantic result

Where subscript $\rightarrow$ represents the $\rightarrow$ of the original program graph created by arrows function on the initial statement. Moreover $\ell_\circ$ and $\ell_\bullet$ have special meaning of being initial and final label respectively. So if we are given a program

$$S_1 \parallel \cdots \parallel S_k$$

and want to construct a program graph, we first calculate the sets $\rightarrow_i$ for each of the $S_i$ (where $1 \leq i \leq k$) using $\text{arrows}[S_i](\ell_\circ, \ell_\bullet)$. And then we can merge all the resulting graphs into one. Note that from this we already have that

$$\forall i : (S_i) \approx_{\rightarrow_i} (\ell_\circ)$$

And also

$$\langle S_1 \parallel \cdots \parallel S_k \rangle \approx_{\rightarrow} \langle \ell_\circ \parallel \cdots \parallel \ell_\circ \rangle$$

So now we want now to prove that whenever a While statement takes a single step in the execution, the program graph representation also takes a step, the resulting state is the same and finally that the destination node is related to the new statement. This amounts to proving that whenever a While statement is executed, the execution of program graph will yield the same result. Note that in the lemma that follows we are only considering statements, not programs that consist of concurrently executing processes.

**Lemma 3.4** If

$$\langle S, \sigma \rangle \xrightarrow{act} \langle S', \sigma' \rangle \quad \text{and} \quad \langle S \rangle \approx_{\rightarrow} \langle \ell \rangle$$

for some $\ell_1$ that

$$\text{arrows}[S](\ell, \ell_1) \subseteq \rightarrow$$

then there exists $\ell'$ such that

$$\langle \ell, \sigma \rangle \xrightarrow{act} \langle \ell', \sigma' \rangle \quad \text{and} \quad \langle S' \rangle \approx_{\rightarrow} \langle \ell' \rangle$$

**Proof.** The proof will proceed by induction.

[skip] Assume

$$\langle \text{skip}, \sigma \rangle \xrightarrow{\text{skip}} \langle \text{done}, \sigma \rangle \quad \text{and} \quad \langle \text{skip} \rangle \approx_{\rightarrow} \langle \ell \rangle$$

We know that for some $\ell'$

$$\{ \ell \xrightarrow{\text{skip}} \ell' \} \subseteq \rightarrow$$
And so
\[ \langle \ell, \sigma \rangle \xrightarrow{\text{skip}} \langle \ell', \sigma \rangle \]

Finally from the definition of \( \approx \rightarrow \)
\[ \langle \text{done} \rangle \approx \rightarrow \langle \ell' \rangle \]

[ass] Assume
\[ \langle x := a, \sigma \rangle \xrightarrow{x := a} \langle \text{done}, \sigma[x \mapsto A[a]\sigma] \rangle \quad \text{and} \quad \langle S \rangle \approx \rightarrow \langle \ell \rangle \]

We also know that for some \( \ell' \)
\[ \{ \ell \xrightarrow{x := a} \ell' \} \subseteq \rightarrow \]

And so
\[ \langle \ell, \sigma \rangle \xrightarrow{x := a} \langle \ell', T[x := a]\sigma \rangle = \langle \ell', \sigma[x \mapsto A[a]\sigma] \rangle \]

Finally from the definition of \( \approx \rightarrow \)
\[ \langle \text{done} \rangle \approx \rightarrow \langle \ell' \rangle \]

[read] Analogously.

[seq1] Assume
\[ \langle S_1; S_2, \sigma \rangle \xrightarrow{\text{act}} \langle S_1'; S_2, \sigma' \rangle \quad \text{and} \quad \langle S_1; S_2 \rangle \approx \rightarrow \langle \ell \rangle \]

because
\[ \langle S_1, \sigma \rangle \xrightarrow{\text{act}} \langle S_1', \sigma' \rangle \]

we also have for some \( \ell_1 \) and \( \ell_2 \)
\[ \text{arrows}[S_1] (\ell_1, \ell_1) \cup \text{arrows}[S_2] (\ell_1, \ell_2) \subseteq \rightarrow \]

We can apply the induction hypothesis on \( \langle S_1, \sigma \rangle \xrightarrow{\text{act}} \langle S_1', \sigma' \rangle \) and \( \langle S_1 \rangle \approx \rightarrow \langle \ell \rangle \) to get
\[ \langle \ell, \sigma \rangle \xrightarrow{\text{act}} \langle \ell', \sigma' \rangle \quad \text{and} \quad \langle S_1' \rangle \approx \rightarrow \langle \ell' \rangle \]

Finally from the above and the definition of \( \approx \rightarrow \)
\[ \langle S_1'; S_2 \rangle \approx \rightarrow \ell' \]
3.4 Semantic result

[seq2] Assume

\[ \langle S_1; S_2, \sigma \rangle \xrightarrow{act} \langle S_2, \sigma' \rangle \quad \text{and} \quad \langle S_1; S_2 \rangle \approx \langle \ell \rangle \]

because

\[ \langle S_1, \sigma \rangle \xrightarrow{act} \langle \text{done}, \sigma' \rangle \]

we also have for some \( \ell_1 \) and \( \ell_2 \)

\[
\text{arrows}[S_1](\ell, \ell_1) \cup \text{arrows}[S_2](\ell_1, \ell_2) \subseteq \]

We can apply the induction hypothesis on \( \langle S_1, \sigma \rangle \xrightarrow{act} \sigma' \) and \( \langle S_1 \rangle \approx \langle \ell \rangle \) to get

\[ \langle \ell, \sigma \rangle \xrightarrow{act} \langle \ell_1, \sigma' \rangle \quad \text{and} \quad \langle \text{done} \rangle \approx \langle \ell_1 \rangle \]

And from the definition of \( \approx \)

\[ \langle S_2 \rangle \approx \ell_1 \]

[if1] Assume

\[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \xrightarrow{b} \langle S_1, \sigma \rangle \]

\[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2 \rangle \approx \langle \ell \rangle \]

because

\[ B[b] = \text{true} \]

we also have for some \( \ell_1, \ell_2 \) and \( \ell_3 \)

\[
\{ \ell \xrightarrow{b} \ell_1, \ell \xrightarrow{\neg b} \ell_2 \} \cup \text{arrows}[S_1](\ell_1, \ell_3) \cup \text{arrows}[S_2](\ell_2, \ell_3) \subseteq \]

From that, the definition of \( T \) and the semantics of program graphs we know that

\[ \langle \ell, \sigma \rangle \xrightarrow{b} \langle \ell_1, \sigma \rangle \]

And from the definition of \( \approx \) we have

\[ \langle S_1 \rangle \approx \langle \ell_1 \rangle \]

[iff] Assume

\[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \xrightarrow{\neg b} \langle S_2, \sigma \rangle \]

\[ \langle \text{if } b \text{ then } S_1 \text{ else } S_2 \rangle \approx \langle \ell \rangle \]

because

\[ B[b] = \text{false} \]

and for some \( \ell_1, \ell_2 \) and \( \ell_3 \) we have

\[
\{ \ell \xrightarrow{b} \ell_1, \ell \xrightarrow{\neg b} \ell_2 \} \cup \text{arrows}[S_1](\ell_1, \ell_3) \cup \text{arrows}[S_2](\ell_2, \ell_3) \subseteq \]

And similarly to the above case we have

\[ \langle \ell, \sigma \rangle \xrightarrow{\neg b} \langle \ell_2, \sigma \rangle \quad \text{and} \quad \langle S_2 \rangle \approx \langle \ell_2 \rangle \]
[while] Assume
\[
\langle \text{while } b \text{ do } S, \sigma \rangle \xrightarrow{b} \langle S; \text{ while } b \text{ do } S, \sigma \rangle
\]
\[
\langle \text{while } b \text{ do } S \rangle \approx \langle \ell \rangle
\]
because
\[
B[b] = \text{true}
\]
We also have for some \( \ell_1 \) and \( \ell_2 \)
\[
\{ \ell \xrightarrow{b} \ell_1, \ell \xrightarrow{b} \ell_2 \} \cup \text{arrows}[S](\ell_1, \ell) \subseteq \rightarrow
\]
Clearly using \( T \) and the definition of \( \approx \rightarrow \) we have
\[
\langle \ell, \sigma \rangle \xrightarrow{b} \langle \ell_1, \sigma \rangle \quad \text{and} \quad \langle S \rangle \approx \langle \ell_1 \rangle
\]
Finally we can conclude
\[
\langle S; \text{ while } b \text{ do } S \rangle \approx \langle \ell_1 \rangle
\]

[whf] Assume
\[
\langle \text{while } b \text{ do } S, \sigma \rangle \xrightarrow{\neg b} \langle \text{done}, \sigma \rangle
\]
\[
\langle \text{while } b \text{ do } S \rangle \approx \langle \ell \rangle
\]
because
\[
B[b] = \text{false}
\]
Moreover for some \( \ell_1 \) and \( \ell_2 \) we have
\[
\{ \ell \xrightarrow{b} \ell_1, \ell \xrightarrow{b} \ell_2 \} \cup \text{arrows}[S](\ell_1, \ell) \subseteq \rightarrow
\]
And from that we can see that
\[
\langle \ell, \sigma \rangle \xrightarrow{\neg b} \langle \ell_2, \sigma \rangle \quad \text{and} \quad \langle \text{done} \rangle \approx \langle \ell_2 \rangle
\]

[ATM] Assume that
\[
\langle \text{atomic}(S), \sigma \rangle \xrightarrow{\text{atomic}(S)} \langle \text{done}, \sigma' \rangle
\]
\[
\langle \text{atomic}(S) \rangle \approx \langle \ell \rangle
\]
because
\[
\langle S, \sigma \rangle \rightarrow^* \langle \text{done}, \sigma' \rangle
\]
and for some \( \ell_1 \) we also have that
\[
\{ \ell \xrightarrow{\text{atomic}(S)} \ell_1 \} \subseteq \rightarrow
\]
Note that $S$ is a sequence of assignment. From the definition of semantics of the WHILE language and $T$ we also know that an assignment will result in the same mapping $\sigma$, thus a sequence of assignments will also have the same result in both cases. So finally

$$\langle \ell, \sigma \rangle \xrightarrow{\text{atomic}(S)} \langle \ell_1, \sigma' \rangle \quad \text{and} \quad \langle \text{done} \rangle \approx \rightarrow \langle \ell \rangle$$

Now with the above lemma we can prove that whenever the program (consisting of multiple concurrently executing statement) takes a single step in execution, then the program graph (that was created from the program) will also proceed one step resulting in the very same state $\sigma$. Moreover the resulting state will be the same and the node will be related to the new program.

**Theorem 3.5** If

$$\langle \cdots \mid S_i \mid \cdots \rangle \approx \rightarrow \langle \cdots, \ell_i, \cdots \rangle$$

and

$$\langle \cdots \mid S_i \mid \cdots, \sigma \rangle \xrightarrow{\text{act}} \langle \cdots \mid S'_i \mid \cdots, \sigma' \rangle$$

because

$$\langle S_i, \sigma \rangle \xrightarrow{\text{act}} \langle S'_i, \sigma' \rangle$$

then

$$\langle \cdots, \ell_i, \cdots, \sigma \rangle \xrightarrow{\text{act}} \langle \cdots, \ell'_i, \cdots, \sigma' \rangle$$

$$\langle \cdots \mid S'_i \mid \cdots \rangle \approx \rightarrow \langle \cdots, \ell'_i, \cdots \rangle$$

**Proof.** From the assumptions we know that

$$\langle S_i, \sigma \rangle \xrightarrow{\text{act}} \langle S'_i, \sigma' \rangle$$

Moreover because

$$\langle \cdots \mid S_i \mid \cdots, \sigma \rangle \xrightarrow{\text{act}} \langle \cdots \mid S'_i \mid \cdots, \sigma' \rangle$$

then from the definition of $\approx \rightarrow$ we have

$$\langle S_i \rangle \approx \rightarrow, \langle \ell \rangle$$

And now we can use the Lemma 3.4 to get

$$\langle \ell_i, \sigma \rangle \xrightarrow{\text{act}} \langle \ell'_i, \sigma' \rangle \quad \text{and} \quad S'_i \approx \rightarrow, \ell'_i$$

From which (and the Definition 3.3 of $\approx \rightarrow$) it follows that

$$\langle \cdots, \ell_i, \cdots, \sigma \rangle \xrightarrow{\text{act}} \langle \cdots, \ell'_i, \cdots, \sigma' \rangle$$

and

$$\langle \cdots \mid S'_i \mid \cdots \rangle \approx \rightarrow \langle \cdots, \ell'_i, \cdots \rangle$$

what finishes the proof.
WHILE language
Chapter 4

Program Graphs and Monotone Framework

4.1 Definitions

Different data flow analyses have quite a lot similarities and it was discovered that they can be expressed in terms of a common framework. It is called Monotone Framework and consists of

- $L$: property space, which is a complete lattice
- $\sqcup$: combination operator
- $F$: finite flow, typically forward or backward
- $E$: extremal labels, typically initial/final labels
- $\iota$: extremal value for extremal labels, $\iota \in L$
- $\mathcal{F}$: set of monotone functions of type $L \to L$
- $f$: mapping from labels to the transfer functions in $\mathcal{F}$

As mentioned in the introduction, the analyses are defined for the entry and exit points of the labeled blocks (here represented with $\circ$ and $\bullet$ subscripts re-
respectively).

\[ \text{Analysis}_\circ(\ell) = \bigsqcup \{ \text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F \} \sqcup \iota_E^\ell \]

where \( \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \)

\[ \text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell)) \]

As this definition should be capable of describing many different analyses, it is fairly high level. Thanks to that and the fact that program graphs are not so much different than flow graphs, only small changes are required to make it work with the new approach.

No change is needed for the property space and the combination operator used to combine the information along different paths, as well as extremal labels. However, the new representation does not have any set \( F \) representing the flow. It uses arrows with some associated actions. Therefore, we should include the set \( \rightarrow \) instead of \( F \). Finally the new representation also requires slightly different transfer functions, however, they will be introduced while discussing the analyses in more detail, as the transfer functions are different for each of them.

\[ L \] property space, which is a complete lattice
\[ \sqcup \] combination operator
\[ \rightarrow \] set of arrows
\[ E \] extremal labels, typically initial/final labels
\[ \iota \] extremal value for extremal labels, \( \iota \in L \)
\[ \mathcal{F} \] set of monotone functions of type \( L \rightarrow L \)
\[ f \] mapping from arrows to the transfer functions in \( \mathcal{F} \)

Obviously in case of the program graphs there are no entry and exit points from labels. They are explicit in the structure of program graphs - an action is an edge connecting two nodes, so we have a label before the action and after it. Another difference is that \( f \) in the original definition is mapping labels to the transfer functions. However, this is not possible in our case, as already mentioned, a single label is not enough to identify an action in a program graph. We solve that by making \( f \) to map arrows to the transfer functions. Therefore, we can
define the analyses in the following way

\[ \text{Analysis}(\ell) = \bigsqcup \{ f_{\ell' \rightarrow \ell}^{\text{act}}(\text{Analysis}(\ell')) \mid \ell' \xrightarrow{\text{act}} \ell \in \cdot \} \sqcup \iota_{E}^{\ell} \]

where \( \iota_{E}^{\ell} = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \)

One interesting thing to notice is that, in program graphs, for any boolean condition (like from an `if` statement) the transfer functions will be used twice - once for the true branch and once for the false one. This is important as some of the analyses can take advantage of this fact. It will be explored more in Chapter 6 that deals with the Constant Propagation analysis.

### 4.2 Worklist algorithm

One of the main advantages of the Monotone Framework is the possibility to define an abstract algorithm, which is able to solve analysis equations for any instance of the framework. One of such generic algorithms is the worklist algorithm - an iterative algorithm that computes the least fixed point solution\(^1\).

**Initialisation:**

\[
\begin{align*}
W & := \text{nil} \\
\text{for all } (\ell, \ell') \text{ in } F & \text{ do} \\
W & := \text{cons}((\ell, \ell'), W) \\
\text{for all } \ell \text{ in } F \text{ or } E & \text{ do} \\
\text{if } \ell \in E & \text{ then Analysis}[\ell] := \iota \\
\text{else Analysis}[\ell] := \bot
\end{align*}
\]

**Iteration:**

\[
\begin{align*}
\text{while } W \neq \text{nil} & \text{ do} \\
(\ell, \ell') & := \text{head}(W) \\
W & := \text{tail}(W) \\
\text{if } f_{\ell}(\text{Analysis}[\ell]) & \not\sqsubseteq \text{Analysis}[\ell'] \text{ then} \\
\text{Analysis}[\ell'] & := \text{Analysis}[\ell'] \sqcup f_{\ell}(\text{Analysis}[\ell]) \\
\text{for all } \ell'' \text{ with } (\ell', \ell'') \in F & \text{ do} \\
W & := \text{cons}(\ell', \ell''), W)
\end{align*}
\]

\(^1\)It is sometimes also called MFP (Maximal Fixed Point) solution.
The results of the analysis can be presented using

\[ MFP_\circ(\ell) := \text{Analysis}[\ell] \]
\[ MFP_\bullet(\ell) := f_\ell(\text{Analysis}[\ell]) \]

The algorithm is quite simple and can be easily transformed to work on the program graphs. The only major difference is using arrows instead of flows as the worklist elements.

**Initialisation:**

\[ W := \text{nil} \]

for all \( \ell \xrightarrow{a} \ell' \) in \( \rightarrow \) do

\[ W := \text{cons}(\ell \xrightarrow{a} \ell', W) \]

for all \( \ell \) in \( F \) or \( E \) do

if \( \ell \in E \)

then \( \text{Analysis}[\ell] := \iota \)

else \( \text{Analysis}[\ell] := \bot_L \)

**Iteration:**

while \( W \neq \text{nil} \) do

\( \ell \xrightarrow{a} \ell' := \text{head}(W) \)

\( W := \text{tail}(W) \)

if \( f_{\ell \xrightarrow{a} \ell'}(\text{Analysis}[\ell]) \nsubseteq \text{Analysis}[\ell'] \) then

\( \text{Analysis}[\ell'] := \text{Analysis}[\ell'] \cup f_{\ell \xrightarrow{a} \ell'}(\text{Analysis}[\ell]) \)

for all \( \ell'' \) with \( \ell' \xrightarrow{a'} \ell'' \) \( \in \rightarrow \) do

\( W := \text{cons}(\ell' \xrightarrow{a'} \ell'', W) \)

So all that was needed to change was the usage of the set \( \rightarrow \) instead of flows \( F \) and the fact that \( f \) maps arrows to transfer functions.

### 4.3 Succinct Solver

#### 4.3.1 About

Succinct Solver is a constraint solver developed by Nielson, Seidl and Nielson [4]. It uses Alternation-free Least Fixed Point Logic (abbreviated ALFP) for the
constraint specification language and is able to compute the least solution for the given set of constraints. Moreover ALFP is a subset of first order logic that is expressive enough to specify analyses such as Reaching Definitions as presented in [5]. Considering that the solver is well optimized and very well tested, it makes perfect sense to use it as the solving engine for different analyses. This makes it possible to focus on the formulation of the analysis itself and not on the solving process. In this chapter we will only introduce the ALFP and Succinct Solver. And then, in the Chapter 5, we will go on and define two different analyses in ALFP and use the solver to obtain the results.

Note that we use here version V2.0 of the Succinct Solver, presented in [6], but it should be well possible to use other versions.

### 4.3.2 ALFP

As already mentioned ALFP is a subset of first order logic and its syntax is presented in the Table 4.1. An ALFP clause is always of the form

\[ cl = cl_1 \land \cdots \land cl_k \]

Moreover it is required that it is, so called, stratified. This basically means that predicates should not depend on the negation of themselves. For more information about it, please consult [4]. Apart from that, the V2.0 of Succinct Solver does not allow universal quantification for preconditions, fortunately this is not necessary for our purposes.

In the next chapter we will present the ALFP clauses generated by the software.

| t ::= \( c \mid x \mid f(t_1,\ldots,t_k) \) |
| pre ::= \( R(t_1,\cdots,t_k) \mid \neg R(t_1,\cdots,t_k) \mid pre_1 \land pre_2 \) |
| \( \mid \) \( pre_1 \lor pre_2 \mid \exists x : pre \mid t_1 = t_2 \mid t_1 \neq t_2 \) |
| \( \mid \forall x : pre \) |
| cl ::= \( R(t_1,\cdots,t_k) \mid 1 \mid cl_1 \land cl_2 \) |
| \( \mid \forall x : cl \mid pre \Rightarrow cl \) |

Table 4.1: ALFP syntax.
and used as the input of the solver, therefore we should also present how the above syntax is expressed in ASCII. The differences are quite small and can be summed up in the Table 4.2.

<table>
<thead>
<tr>
<th>∀x : ··· A x. ···</th>
<th>∃x : ··· E x. ···</th>
</tr>
</thead>
<tbody>
<tr>
<td>··· ⇒ ··· ··· ⇒  ···</td>
<td>··· ∧ ··· ··· &amp;  ···</td>
</tr>
<tr>
<td>··· ∨ ··· ···</td>
<td>···</td>
</tr>
</tbody>
</table>

Table 4.2: ASCII representation of ALFP.
Chapter 5

Reaching Definitions and Live Variables

5.1 Reaching Definitions

5.1.1 Introduction

With the adjustments to the Monotone Framework, we can go on and actually define some analyses. First of all let us have a look at one of the classical ones, namely Reaching Definitions. The analysis tries to determine (quote from [3]):

*For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.*

So basically the analysis keeps track of all the performed assignments that have not been overwritten and includes all the possibilities from different paths (for
instance assignments in different branches of the if conditional). One of the possible applications of RD is in so called Program Slicing, which aims at indicating what parts of the program might have influenced the values at some point of interest. Therefore it is quite often used in debugging.

5.1.2 Classical formulation

To begin with, let us consider the original definition of the analysis that works on flow graphs. Recall that the analysis is interested in tracking the assignment that have not been overwritten. Therefore, the elements of the lattice consist only of a variable identifier and a label of the assignment

\[ L = \mathcal{P}(\text{Var}_* \times \text{Lab}_*) \]

As the analysis is a may analysis, i.e. keeps track of all possible assignments, it uses \( \cup \) as the combination operator and subset inclusion \( \subseteq \) for the ordering. Apart from that, it is a forward analysis and starts from the initial block.

\[
\text{RD}_{\text{entry}}(\ell) = \begin{cases} 
\{ (x,?) \mid x \in \text{Var}_* \} & \text{if } \ell = \text{init}(S_*) \\
\cup \{ \text{RD}_{\text{exit}}(\ell') \mid (\ell', \ell) \in \text{flow}(S_*) \} & \text{otherwise}
\end{cases}
\]

\[
\text{RD}_{\text{exit}}(\ell) = f_\ell(\text{RD}_{\text{entry}}(\ell))
\]

The analysis works by analysing each block and checking whether some previous assignments have been killed and a new one has been generated. So the transfer function is defined in terms of the functions gen and kill

\[
f_\ell(l) = (l \setminus \text{kill}([B]^\ell)) \cup \text{gen}([B]^\ell)
\]

where \( l \in L \) and \([B]^\ell \in \text{Blocks}. And the set of transfer functions for RD can be expressed as

\[
\mathcal{F} = \{ f : L \rightarrow L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g \}
\]

where \( L \) is the property space of the analysis and \( l, l_k, l_g \in L \). Finally, the gen\textsubscript{RD} and kill\textsubscript{RD} functions are only concerned with the blocks that assign to a variable and can be defined as

\[
\text{kill}_{\text{RD}}(\ell) = \{ (x,?) \} \cup \{ (x, \ell') \mid [B]^\ell \text{ assigns to } x \text{ in } S_* \}
\]

\[
\text{gen}_{\text{RD}}(\ell) = \{ (x, \ell) \}
\]

For all the other blocks kill\textsubscript{RD} and gen\textsubscript{RD} return empty sets.
Before we go on, let us have a look at an example that presents how the analysis works. Consider the program

\[
\text{read } x; \text{ if } x > 0 \text{ then } y := 1 \text{ else } (x := 0; y := 0); z := y
\]

The flow graph for the program is presented in the Figure 5.1. We will not go on through the whole process of solving all the equations, but will just present the results. Note that for such a simple program the results are quite obvious and easy to understand. Results for the entry points are

\[
\begin{align*}
\text{Analysis}_0(1) &= \{(x,?), (y,?), (z,?)\} \\
\text{Analysis}_0(2) &= \{(x,1), (y,?), (z,?)\} \\
\text{Analysis}_0(3) &= \{(x,1), (y,?), (z,?)\} \\
\text{Analysis}_0(4) &= \{(x,1), (y,?), (z,?)\} \\
\text{Analysis}_0(5) &= \{(x,4), (y,?), (z,?)\} \\
\text{Analysis}_0(6) &= \{(x,1), (x,4), (y,3), (y,5), (z,?)\}
\end{align*}
\]
And for the exit of labels we have

\[
\begin{align*}
Analysis_\bullet(1) &= \{(x, 1), (y, ?), (z, ?)\} \\
Analysis_\bullet(2) &= \{(x, 1), (y, ?), (z, ?)\} \\
Analysis_\bullet(3) &= \{(x, 1), (y, 3), (z, ?)\} \\
Analysis_\bullet(4) &= \{(x, 4), (y, ?), (z, ?)\} \\
Analysis_\bullet(5) &= \{(x, 4), (y, 5), (z, ?)\} \\
Analysis_\bullet(6) &= \{(x, 1), (x, 4), (y, 3), (y, 5), (z, 6)\}
\end{align*}
\]

Observe that the analysis combined the information from both branches of the if statement in the way that preserves all the possibilities.

### 5.1.3 Program Graph version

Only little changes are required to make the analysis work on a program graphs. There are only two major differences. First of all we are interested in the results of analysis for the nodes (i.e. labels), not the entry and exit points of blocks. Secondly we use the information in $\rightarrow$ for how the control might flow in a program. And so the analysis can be defined as follows:

\[
RD(\ell) = \begin{cases} 
\{(x, ?) \mid x \in Vars\} & \text{if } \ell = Lab_0 \\
\bigcup \{ (RD(\ell')) \setminus kill_{RD}(arr)) \cup gen_{RD}(arr) \\
& \mid arr = \ell' \xrightarrow{act} \ell, \ arr \in \rightarrow \} & \text{otherwise}
\end{cases}
\]

Another change is how the results are created and interpreted. In the original approach, the pair of a variable identifier and a label identified exactly at what block has the variable been assigned. This is not exactly possible when an action cannot be identified by a single label. However, we might as well simply treat the label as identifying the node at which the action originated, i.e. the source of the edge making the assignment. And so the definitions of $gen_{RD}$ and $kill_{RD}$ using the new approach are quite similar to the previous ones.

\[
\begin{align*}
kill_{RD}(\ell \xrightarrow{x \leftarrow a} \ell') &= \{(x, \ell_k) \mid \ell_k \xrightarrow{a'} \ell'_k \in \rightarrow, \ a' \text{ assigns to } x\} \\
kill_{RD}(\ell \xrightarrow{\text{read } x} \ell') &= \{(x, \ell) \mid \ell \in \rightarrow, \ x \text{ is read}\} \\
gen_{RD}(\ell \xrightarrow{x \leftarrow a} \ell') &= \{(x, \ell) \} \\
gen_{RD}(\ell \xrightarrow{\text{read } x} \ell') &= \{(x, \ell) \}
\end{align*}
\]
5.1 Reaching Definitions

In case of other actions, both of the functions return empty sets, just like the original definitions.

The results of the analysis of the example program (from the previous section) are presented on the Figure 5.2. Note that this is the actual output of the tool implemented as part of the thesis. Apart from a little bit different labels, the results are basically the same as in the case of the original analysis. Note that the implementation uses 0 instead of "?" for the uninitialised variables.
5.1.4 ALFP formulation

It is actually quite easy to specify the RD analysis using the ALFP. First of all it is useful to create predicates such as

\[
\begin{align*}
Label &= \text{the term is a label} \\
Init &= \text{the term is an initial label} \\
Var &= \text{the term is a variable} \\
Edge &= \text{the terms correspond to an edge}
\end{align*}
\]

that can be then used in the main clauses. From section 5.1.2 we know that RD is defined in the following way

\[
RD(\ell) = \begin{cases}
\{ (x, ?) \mid x \in \mathbf{Var}, \} & \text{if } \ell \in \mathbf{Lab}_o \\
\bigcup \{ \text{RD}(\ell') \setminus \text{kill}_{\text{RD}}(\text{arr}) \cup \text{gen}_{\text{RD}}(\text{arr}) \mid \text{arr} = \ell' \xrightarrow{\text{act}} \ell, \text{arr} \in \rightarrow \} & \text{otherwise}
\end{cases}
\]

Clearly we also need to create 4-ary predicates \textit{Kill} and \textit{Gen}, that will be generated by the software analysing the program for the right actions. The fist two terms of both of these predicates correspond to the edge, and the second two to the variable and label at which the variable has been defined. Finally we also need to introduce the ternary predicate \textit{Analysis}, where the first term is indicating to which label is the result associated with, the second is the variable being defined and the third is the label where the definition occurred. All the \textit{Gen} predicates can easily be generated by the software analysing the program. It is a bit more complex with the \textit{Kill} ones, as they should take into account all other assignments to the given variable. One of the ways to handle that is to use the results of the analysis at the node from which the edge with assignment/read action originates. If there is any information regarding a definition of some variable then it can be killed. For instance if there is an assignment to variable \(x\) on the edge from the node 1 to 2, we have

\[
\forall \ell : Label(\ell) \land Analysis(1, x, \ell) \Rightarrow Kill(1, 2, x, \ell)
\]

And so the solver will do most of the work itself, including what variable-label pairs should be killed. Note that such clause must be generated for every assignment/read statement. The last thing are clauses that would express when the \textit{Analysis} predicate would be used. First of all, the case of the initial label can be described as

\[
\forall \ell, v : Init(\ell) \land Var(v) \Rightarrow Analysis(l, v, 0)
\]
And for all other nodes we have one general clause

\[ \forall \text{from}, \text{to}, l, v : \text{Edge}(\text{from}, \text{to}) \Rightarrow \]
\[ (\text{Analysis}(\text{from}, v, l) \land \neg \text{Kill}(\text{from}, \text{to}, v, l)) \lor \text{Gen}(\text{from}, \text{to}, v, l) \]
\[ \Rightarrow \text{Analysis}(\text{to}, v, l) \]

Which basically says that if the variable-label pair was at the previous label and it has not been killed, or alternatively it has been generated on the current edge, then it should be included in the analysis results at the current node.

To make it easier and more understandable, let us have a look at our example program (repeated here for the reader’s convenience).

```
read x; if x > 0 then y := 1 else (x := 0; y := 0); z := y
```

Based on the above considerations, the ALFP constraints that have been generated for this example by our tool are as follows

\[
\text{Label}(I_0) \land \text{Label}(I_1) \land \text{Label}(I_2) \land \text{Label}(I_3) \land \text{Label}(I_4) \land \text{Label}(I_5) \land \text{Label}(I_6) \land \text{Label}(I_7) \land \text{Init}(I_1) \land \text{Final}(I_7) \land \text{Var}(x) \land \text{Var}(y) \land \text{Var}(z) \land \text{Edge}(I_1, I_2) \land \text{Edge}(I_2, I_5) \land \text{Edge}(I_2, I_3) \land \text{Edge}(I_3, I_4) \land \text{Edge}(I_4, I_7) \land \text{Edge}(I_5, I_6) \land \text{Edge}(I_6, I_4) \land \text{Gen}(I_1, I_2, x, I_1) \land (A l. ((\text{Label}(l) \land \text{Analysis}(I_1, x, l)) \Rightarrow \text{Kill}(I_1, I_2, x, l))) \land \text{Gen}(I_3, I_4, y, I_3) \land (A l. ((\text{Label}(l) \land \text{Analysis}(I_3, y, l)) \Rightarrow \text{Kill}(I_3, I_4, y, l))) \land \text{Gen}(I_4, I_7, z, I_4) \land (A l. ((\text{Label}(l) \land \text{Analysis}(I_4, z, l)) \Rightarrow \text{Kill}(I_4, I_7, z, l))) \land \text{Gen}(I_5, I_6, x, I_5) \land (A l. ((\text{Label}(l) \land \text{Analysis}(I_5, x, l)) \Rightarrow \text{Kill}(I_5, I_6, x, l))) \land \text{Gen}(I_6, I_4, y, I_6) \land (A l. ((\text{Label}(l) \land \text{Analysis}(I_6, y, l)) \Rightarrow \text{Kill}(I_6, I_4, y, l))) \land (A l. (A v. ((\text{Init}(l) \land \text{Var}(v)) \Rightarrow \text{Analysis}(l, v, I_0)))) \land (A l f. (A l t. (A l. (A v. (\text{Edge}(l f, l t) \Rightarrow \text{(Analysis}(l f, v, l) \land \neg \text{Kill}(l f, l t, v, l)) \lor \text{Gen}(l f, l t, v, l)) \Rightarrow \text{Analysis}(l t, v, l)))))))
\]

Finally the solution is obtained by solving the above formula by the Succinct Solver and then interpreting its output by the developed software. The result
in the form of a program graph is available in the Figure 5.3.

Figure 5.3: Results of the RD analysis using the Succinct Solver.
5.2 Live Variables

5.2.1 Definitions

Another quite well-known analysis is the Live Variables analysis. It simply determines what variables may be live at the given point in the program. Where live is meant that the value of the variable might be actually used before being overwritten by another assignment. Consider for example the program

\[
x := 1; \quad y := 1; \quad x := y; \quad y := 2
\]

Clearly \(x\) is not used between the first and the second assignment, thus it is not live there. This is quite different in case of the \(y\) variable. After the first assignment, its value is actually used, and only then overwritten. Therefore, before the second assignment to \(x\), \(y\) is live.

LV is also an instance of the Monotone Framework and works on a sets of variables (that are live at the given point). It also uses the union of sets for the combination operator and subset inclusion for ordering. It is usually defined in the following way (i.e. the version using flow graphs).

\[
LV_{exit}(\ell) = \begin{cases} 
\emptyset & \text{if } \ell \in \text{final}(S) \\
\bigcup \{ (LV_{entry}(\ell') \mid (\ell', \ell) \in \text{flow}^R(S)) \} & \text{otherwise}
\end{cases}
\]

\[
LV_{entry}(\ell) = ((LV_{exit}(\ell) \setminus \text{kill}_{LV}([B]^{\ell})) \cup \text{gen}_{LV}([B]^{\ell})
\]

And again the kill and gen functions return \(\emptyset\) for all blocks other than the listed below.

\[
\text{kill}_{LV}([x := a]^{\ell}) = \{x\}
\]

\[
\text{kill}_{LV}([\text{read } x]^{\ell}) = \{x\}
\]

\[
\text{gen}_{LV}([x := a]^{\ell}) = \text{FV}(a)
\]

\[
\text{gen}_{LV}([b]^{\ell}) = \text{FV}(b)
\]

The \(\text{FV}\) function is simply returning a set of free variables used in the given expression.

One of the first things to notice is that LV is a backward analysis. It starts from the final labels and then continues using the flow\(^R\) (reverse flow). This is an
important consideration when redefining it to work with the program graphs. And so we have to define the \( \text{LV}(\ell) \) in terms of \( \text{LV}(\ell') \) where there is an edge from \( \ell \) to \( \ell' \) and not the other way round as in RD.

\[
\text{LV}(\ell) = \begin{cases} 
\emptyset & \text{if } \ell \in \text{Lab}. \\
\bigcup \{ (\text{LV}(\ell') \setminus \text{kill}_{\text{LV}}(\text{arr})) \cup \text{gen}_{\text{LV}}(\text{arr}) 
| \text{arr} = \ell \xrightarrow{\text{act.}} \ell', \text{arr} \in \rightarrow \} & \text{otherwise}
\end{cases}
\]

The above definition expresses that and if compared with the RD one, it clearly works backwards. And finally the \( \text{kill}_{\text{LV}} \) and \( \text{gen}_{\text{LV}} \) functions

\[
\begin{align*}
\text{kill}_{\text{LV}}(\ell \ x := a \xrightarrow{} \ell') &= \{x\} \\
\text{kill}_{\text{LV}}(\ell \ \text{read } x \xrightarrow{} \ell') &= \{x\} \\
\text{gen}_{\text{LV}}(\ell \ x := a \xrightarrow{} \ell') &= \text{FV}(a) \\
\text{gen}_{\text{LV}}(\ell \ b \xrightarrow{} \ell') &= \text{FV}(b)
\end{align*}
\]

Let us have again a look at an example that would demonstrate the analysis.

```
read x; read y; z := 10; if x>0 then z:=x-1 else z:=y+1; x:=z
```

The result of the analysis is presented on the Figure 5.4. This example presents some of the properties of the analysis. For instance \( y \) is live before the if conditional even though it is used only in one of the branches. Also \( z \) is not live at nodes 1, 2, 3, 4, 5 and 7 as its value is overwritten no matter which branch is chosen. This also demonstrates that Live Variables can be used for the optimization purposes - the value of \( z \) after the first assignment is not used before being overwritten. Therefore, this assignment could be completely removed without changing the meaning of the program.

### 5.2.2 ALFP formulation

The ALFP formulation of the LV analysis is actually a little bit simpler than the one for RD. The main reason for this is that it is concerned only with variable names, without associated labels. And this much simplifies the issue with the \text{Kill} predicate, as we do not have to handle the possible values of these additional labels. Also we have to pay attention how to define the formula for the \text{Analysis} predicate, as this is a backward analysis. Finally the value for
extremal labels (i.e. here final label) is empty set, so we do not have to actually do any initialisation step.

Apart from that, as before the implementation is responsible for generating all the \textit{Gen} and \textit{Kill} predicates. And as before they are associated with the right edge, so \textit{Gen}(1, 2, x) would mean that variable \(x\) is used on the edge from node 1 to node 2. As for the global formula expressing how the results at different nodes are related, it can be defined as

\[
\forall \text{from, to, } v : \text{Edge(from, to)} \Rightarrow \\
(\text{Analysis(to, v)} \land \neg \text{Kill(from, to, v)}) \lor \text{Gen(from, to, v)} \\
\Rightarrow \text{Analysis(from, v)}
\]

For the example program presented in the previous section, the following constraints have been generated.

\[
\text{Label(I0)} \land \text{Label(I1)} \land \text{Label(I2)} \land \text{Label(I3)} \land \\
\text{Label(I4)} \land \text{Label(I5)} \land \text{Label(I6)} \land \text{Label(I7)} \land \\
\text{Label(I8)} \land \text{Init(I1)} \land \text{Final(I8)} \land \\
\text{Var(x)} \land \text{Var(y)} \land \text{Var(z)} \land \\
\text{Edge(I1, I2)} \land \text{Edge(I2, I3)} \land \text{Edge(I3, I4)} \land \\
\text{Edge(I4, I7)} \land \text{Edge(I4, I5)} \land \text{Edge(I5, I6)} \land \\
\text{Edge(I6, I8)} \land \text{Edge(I7, I6)} \land \\
\text{Kill(I1, I2, x)} \land \\
\text{Kill(I2, I3, y)} \land \\
\text{Kill(I3, I4, z)} \land \\
\text{Gen(I4, I7, x)} \land \\
\text{Gen(I4, I5, x)} \land \\
\text{Gen(I5, I6, x)} \land \\
\text{Kill(I5, I6, z)} \land \\
\text{Gen(I6, I8, z)} \land \\
\text{Kill(I6, I8, x)} \land \\
\text{Gen(I7, I6, y)} \land \\
\text{Kill(I7, I6, z)} \land \\
(A \text{ lf.} (A \text{ lt.} (A \text{ v.} (\text{Edge(lf, lt)} \Rightarrow \\
(\text{((Analysis(lt, v)} \land \neg \text{Kill(lf, lt, v)}) \\
\lor \text{Gen(lf, lt, v)})) \Rightarrow \text{Analysis(lf, v)}))))))
\]

The results are precisely the same as in the Figure 5.4.
Figure 5.4: Results of the LV analysis.
Chapter 6

Constant Propagation

6.1 Independent attribute Constant Propagation

Constant Propagation is an analysis that checks what variables might be constant at any given point in the program. Obviously it is often used in compilers for optimisation purposes. CP is also an instance of the Monotone Framework, but a bit more complex than the previously mentioned analyses. For instance it cannot be expressed using only gen and kill functions. Again we are basing the presentation of the analysis on [3]. The lattice that the analysis works on is defined by

\[ \text{State}_{CP} = ((\text{Var}, \to Z^\top), \bot, \sqcap, \sqcup, \sqcap, \bot, \lambda x. \top) \]

The elements of the lattice are mappings from the variable identifiers present in the program (\(\text{Var},\)) to \(Z^\top = Z \cup \{\top\}\), that is their value (integers) or \(\top\). We use \(\top\) to express that a variable might not be constant at the given point. \(Z^\top\)
\( A_{\text{CP}}[n] \hat{\sigma} = \begin{cases} \bot & \text{if } \hat{\sigma} = \bot \\ n & \text{otherwise} \end{cases} \)

\( A_{\text{CP}}[x] \hat{\sigma} = \begin{cases} \bot & \text{if } \hat{\sigma} = \bot \\ \hat{\sigma}(x) & \text{otherwise} \end{cases} \)

\( A_{\text{CP}}[a_1 \; \textit{op}_a \; a_2] \hat{\sigma} = A_{\text{CP}}[a_1] \hat{\sigma} \; \textit{op}_a \; A_{\text{CP}}[a_2] \hat{\sigma} \)

Table 6.1: Arithmetic expressions for CP.

is partially ordered in the following way

\[
\forall z \in \mathbb{Z}^\top : z \sqsubseteq_z \top \\
\forall z_1, z_2 \in \mathbb{Z}^\top : (z_1 \sqsubseteq_z z_2) \iff (z_1 = z_2)
\]

The lattice itself \((\texttt{Var}_* \rightarrow \mathbb{Z}^\top)_\bot\) is partially ordered by \(\sqsubseteq_{\text{CP}}\)

\[
\forall \hat{\sigma} \in (\texttt{Var}_* \rightarrow \mathbb{Z}^\top)_\bot : \bot \sqsubseteq_{\text{CP}} \hat{\sigma} \\
\forall \hat{\sigma} \in \texttt{Var}_* \rightarrow \mathbb{Z}^\top : \hat{\sigma}_1 \sqsubseteq_{\text{CP}} \hat{\sigma}_2 \iff \forall x : \hat{\sigma}_1(x) \sqsubseteq \hat{\sigma}_2(x)
\]

We also define functions working with arithmetic and boolean expressions using \(\hat{\sigma}\). They are presented in Tables 6.1 and 6.2 respectively. We assume that we also have the \(\textit{op}_a, \textit{op}_b\) and \(\textit{op}_r\) that are simply versions of \(\textit{op}_a, \textit{op}_b\) and \(\textit{op}_r\), lifted from \(\mathbb{Z}\) to \(\mathbb{Z}_\bot^\top\). So for instance, let \(z_1, z_2 \in \mathbb{Z}_\bot^\top\)

\[
z_1 \; \textit{op}_a \; z_2 = \begin{cases} z_1 \; \textit{op}_a \; z_2 & \text{if } z_1, z_2 \in \mathbb{Z} \\
\bot & \text{if } z_1 = \bot \lor z_2 = \bot \\
\top & \text{otherwise} \end{cases}
\]

The definition of transfer functions for the Constant Propagation working on flow graphs is presented in the Table 6.3. The functions simply try to record
what values are assigned to different variables. And so neither a conditional or skip statement change anything. The interesting statements are naturally the ones that actually assign values to the variables.

The adjustments (and extensions for our version of WHILE) to make the transfer functions work with program graphs are again quite small and are presented in the Table 6.4. As mentioned in the introduction, the fact that each condition is translated into two edges, can be used to make the analysis a little bit more precise. The concept can be illustrated with a very simple program

read x; if x == 0 then x := x + 1 else x := 1; y := x

The original definition would simply return that y would be \( \top \) because x is \( \top \) after the read statement, and then either is incremented or set to 0, but the analysis has to be safe and thus assume that x is not constant. On the other hand it is clear that the value of y will be 1 - if x is 0 then it is incremented, otherwise it is set to 1. The end result: x is always equal to 1 after the if statement. The original definition of CP does not "know" which of the flows corresponds to the condition being true or false. But with program graphs we have two separate edges and when the analysis knows that x is \( \top \) and gets to the edge with x == 0, it can conclude that the only possible value x can have, so that the condition is true, is 0. And after that the rest is straightforward. Note that this works best in special cases where the conditions test for equality. In the situation when a condition is using smaller/greater than operators, the

Table 6.2: Boolean expressions for CP.
\[ \text{Table 6.3: Transfer functions for CP using flow graphs.} \]

\[ \text{Table 6.4: Transfer functions for CP using program graphs.} \]
analysis probably will not be improved - there might be many values of the variable(s) for which such a condition would be true and taking the least upper bound will map such variables to $\top$.

As a side note, it should be mentioned that, the current implementation of handling the conditions is much less powerful than what is presented above. It simply tries to substitute the variables to their values (if they are constant), evaluate the expressions (if possible) and see whether there is a equality test of a variable with a constant. If that is the case, it is recorded and if that was the only value for this variable, then it must be the case that the variable has this value. Therefore the implementation will not be able to handle complex expressions, but will still be more precise in some simple scenarios.

However, even with the more powerful version of handling conditions, the analysis is still not precise enough in many situations. For

```
read x; if x > 0 then (x := 1; y := 1) else (x := 0; y := 0)
```

the analysis result is that both of the variables are mapped to $\top$. And this completely fails to capture the fact that there are only two possible configurations - either both of the variables are 1 or both of them are 0. This is also why it is called independent attribute analysis - it does not maintain any relations between the variables. One of the solutions for this problem is to use the relational analysis, which is the focus of the next section.

## 6.2 Relational Constant Propagation

### 6.2.1 Introduction

First of all let us consider what is required and desired for a relational Constant Propagation analysis. Obviously we need a different lattice that would be able to store the information about many possible mappings (such as those used in the original CP). One possibility is to simply use sets $\mathcal{L}$ such that $\mathcal{L} \subseteq \mathcal{P}(\text{Var} \to \mathbb{Z}^+)$. There is one major challenge associated with using such sets. Even if we consider that every program has a finite number of variables, we map them to integers and there is an infinite number of them, and so there is potentially
infinite number of mappings. For an example consider the program

\[
\text{x := 1; while x > 1; do x := x + 1}
\]

The number of possible values of x is infinite and we have to make sure that our analysis will not try to find all of them (i.e. will not loop). Therefore we need to somehow ensure that there is a way to limit the amount of information/size of the sets.

Apart from that we need to have a partial ordering of those sets. However, in our situation it is not possible to use the \(\subseteq\). There are basically two problems with it, firstly it does not take into account the fact that some of the mappings might be more general (larger or equal by the ordering) than other. For example consider the following

\[
\begin{array}{l}
\{[x \mapsto 1], [x \mapsto 2]\} \not\subseteq \{[x \mapsto \top]\}
\end{array}
\]

Clearly we would like the second set to be somehow larger than the first one - it is more general and means that \(x\) can in fact have any value and that obviously includes both 1 and 2. Secondly, consider that we want to take an upper bound of two sets such as \(L_1 \not\subseteq L_2\) and \(L_2 \not\subseteq L_1\). Whatever upper bound operator we would use (least upper bound operator or some widening one) the size of the resulting set will be greater than that of either \(L_1\) or \(L_2\). So there is no way to limit the size of the resulting set. And this means that will not be able to handle programs such as mentioned in the above paragraph.

Before we go on and actually define the ordering \(\subseteq_{\text{RCP}}\) that would avoid the above problems, let us have a look at a few examples on how we would like it to work. For simplicity the examples will use sets of one-, or two-element mappings. First of all we clearly want to be able to add some information to the set of possible mappings, for instance

\[
\{[x \mapsto 1, y \mapsto 2]\} \subseteq_{\text{RCP}} \{[x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 2]\}
\]

As already mentioned, we would also like to have

\[
\{[x \mapsto 1, y \mapsto 10], [x \mapsto 2, y \mapsto 10]\} \subseteq_{\text{RCP}} \{[x \mapsto \top, y \mapsto 10]\}
\]

But

\[
\{[x \mapsto 1, y \mapsto 10], [x \mapsto 2, y \mapsto 10]\} \not\subseteq_{\text{RCP}} \{[x \mapsto \top, y \mapsto 20]\}
\]

In other words, if we have \(L_1 \subseteq_{\text{RCP}} L_2\), then for every mapping \(\hat{\sigma}_1\) in the first set, it should be the case that either:

- it is included in the second one as well, or
• there is a more general mapping $\hat{\sigma}_2$, $\hat{\sigma}_1 \preceq_{\text{cp}} \hat{\sigma}_2$ in the second one.

This would ensure that no information about possible values of variables is lost.

Another interesting thing is the fact that if we have a set with mappings, it is the most general ones that are really important. Consider

$$\{[x \mapsto 1, y \mapsto 10], [x \mapsto \top, y \mapsto 10]\}$$

Because the $y$ variable is 10 in both mappings, the first one does not really provide anything that is not included in the second one. This is also expressed by the fact that according to the definition of $\subseteq_{\text{cp}}$ we have

$$[x \mapsto 1, y \mapsto 10] \subseteq_{\text{cp}} [x \mapsto \top, y \mapsto 10]$$

So removing the first mapping does not really hurt the precision of the analysis and we could as well work on a smaller set $\{[x \mapsto \top, y \mapsto 10]\}$.

### 6.2.2 Partial order

With the above, we have some idea of what we would like the partial ordering to look like. However, there are still some issues that need to be resolved, for instance let

$$\mathcal{L}_1 = \{[x \mapsto 1, y \mapsto 10], [x \mapsto 2, y \mapsto 10], [x \mapsto \top, y \mapsto 10]\}$$

$$\mathcal{L}_2 = \{[x \mapsto 1, y \mapsto 10], [x \mapsto \top, y \mapsto 10]\}$$

Now which one should be larger/smaller than the other? From the point of view of the analysis, they basically express the same result. Whatever the decision, it can have profound consequences for the definition of the ordering and make it quite a bit more complex. However, we can avoid choosing one of them and all the associated consequences, by restricting the sets that the analysis would work on.

**Definition 6.1** The elements of lattice for relational constant propagation are members of the set

$$\text{State}_{\text{RCP}} = \{\mathcal{L} \mid \mathcal{L} \in \mathcal{P}(\text{Var}_* \rightarrow \mathbb{Z}^\top), \forall \hat{\sigma}, \hat{\sigma}' \in \mathcal{L} : \hat{\sigma} \subseteq_{\text{cp}} \hat{\sigma}' \Rightarrow \hat{\sigma} = \hat{\sigma}'\}$$

This restricts the sets to the ones that only contain mappings that are not comparable, i.e. for every two different mappings in a set, neither of them can
be larger/smaller by $\subseteq_{\text{RCP}}$. As already mentioned this does not hurt the precision of the analysis and will be quite useful in the definition of partial order and in the accompanying proof.

**Definition 6.2** We define partial ordering $\subseteq_{\text{RCP}}$ that will work on the above defined sets as follows

$$L_1 \subseteq_{\text{RCP}} L_2 \quad \text{iff} \quad \forall \hat{\sigma}_1 \in L_1 \ \exists \hat{\sigma}_2 \in L_2 : \hat{\sigma}_1 \subseteq_{\text{CP}} \hat{\sigma}_2$$

This expresses that $L_2$ is required to include all the possible values of variables that $L_1$ has. Which is exactly what we have already mentioned as in the previous section.

Now all we have to prove is that this is in fact a partial order. Let us start with the following two Lemmas proving that $\subseteq_{\text{RCP}}$ is transitive and anti-symmetric.

**Lemma 6.3** The relation $\subseteq_{\text{RCP}}$ is transitive.

**Proof.** We have to prove that for any $L_1, L_2, L_3 \in \mathcal{State}_{\text{RCP}}$ we have:

$$L_1 \subseteq_{\text{RCP}} L_2 \land L_2 \subseteq_{\text{RCP}} L_3 \Rightarrow L_1 \subseteq_{\text{RCP}} L_3$$

From the definition of $\mathcal{State}_{\text{RCP}}$ and $\subseteq_{\text{RCP}}$ we know that

$$\forall \hat{\sigma}_1 \in L_1 \ \exists \hat{\sigma}_2 \in L_2 : \hat{\sigma}_1 \subseteq_{\text{CP}} \hat{\sigma}_2$$

and

$$\forall \hat{\sigma}_2 \in L_2 \ \exists \hat{\sigma}_3 \in L_3 : \hat{\sigma}_2 \subseteq_{\text{CP}} \hat{\sigma}_3$$

So we know that for each of the mappings in $L_1$ there exists a larger one in $L_2$, for which there exists yet larger one in $L_3$. So from the transitivity of $\subseteq_{\text{CP}}$ we can conclude that

$$\forall \hat{\sigma}_1 \in L_1 \ \exists \hat{\sigma}_3 \in L_3 : \hat{\sigma}_1 \subseteq_{\text{CP}} \hat{\sigma}_3$$

which gives $L_1 \subseteq_{\text{RCP}} L_3$ as required.

**Lemma 6.4** The relation $\subseteq_{\text{RCP}}$ is anti-symmetric.

**Proof.** We have to prove that for any $L_1, L_2 \in \mathcal{State}_{\text{RCP}}$ we have

$$L_1 \subseteq_{\text{RCP}} L_2 \land L_2 \subseteq_{\text{RCP}} L_1 \Rightarrow L_1 = L_2$$
From the definition of $\hat{\text{State}}_{RCP}$ and $\sqsubseteq_{RCP}$ we know:

$$
\forall \hat{\sigma}_1 \in L_1 \quad \exists \hat{\sigma}_2 \in L_2 : \hat{\sigma}_1 \sqsubseteq_{cp} \hat{\sigma}_2 \\
\forall \hat{\sigma}_2 \in L_2 \quad \exists \hat{\sigma}_1 \in L_1 : \hat{\sigma}_2 \sqsubseteq_{cp} \hat{\sigma}_1 \\
\forall \hat{\sigma}_1, \hat{\sigma}_1' \in L_1 : \hat{\sigma}_1 \sqsubseteq_{cp} \hat{\sigma}_1' \Rightarrow \hat{\sigma}_1' = \hat{\sigma}_1 \\
\forall \hat{\sigma}_2, \hat{\sigma}_2' \in L_2 : \hat{\sigma}_2 \sqsubseteq_{cp} \hat{\sigma}_2' \Rightarrow \hat{\sigma}_2' = \hat{\sigma}_2
$$

From the above we can argue that for every $\hat{\sigma}_1 \in L_1$ there exists $\hat{\sigma}_2 \in L_2$ such that $\hat{\sigma}_1 \sqsubseteq_{cp} \hat{\sigma}_2$. Moreover there should also exist $\hat{\sigma}_1' \in L_1$ such that $\hat{\sigma}_2 \sqsubseteq_{cp} \hat{\sigma}_1'$. But then we have:

$$
\hat{\sigma}_1 \sqsubseteq_{cp} \hat{\sigma}_2 \sqsubseteq_{cp} \hat{\sigma}_1'
$$

And as $\sqsubseteq_{cp}$ is anti-symmetric itself, we have $\hat{\sigma}_1 = \hat{\sigma}_2$. This means that each element in $L_1$ is also in $L_2$. However, we can perform the above reasoning also the other way round and prove that each element in $L_2$ is in $L_1$. Therefore $L_1 = L_2$.

**Theorem 6.5** The set $\hat{\text{State}}_{RCP}$ is partially ordered by $\sqsubseteq_{RCP}$.

**PROOF.** It is immediate that $\sqsubseteq_{RCP}$ is reflexive. From Lemmas 6.3 and 6.4 it follows that it is also transitive and anti-symmetric. Therefore $\hat{\text{State}}_{RCP}$ is partially ordered by $\sqsubseteq_{RCP}$.

### 6.2.3 Lattice

Having a partially ordered set we can now go on to define the lattice that will be used by the relational CP. We start with the definition of the least upper bound operator.

**Definition 6.6** Let $\mathcal{Y}$ be a subset of $\hat{\text{State}}_{RCP}$, we define the $\bigcup_{RCP} \mathcal{Y}$ as

$$
\bigcup_{RCP} \mathcal{Y} = \{ \hat{\sigma} \mid \hat{\sigma} \in \bigcup \mathcal{Y}, \quad \forall \hat{\sigma}' \in \bigcup \mathcal{Y} : \hat{\sigma} \sqsubseteq_{cp} \hat{\sigma}' \Rightarrow \hat{\sigma} = \hat{\sigma}' \}
$$

This amounts to choosing only such mappings from $\bigcup \mathcal{Y}$ that are not smaller than any other elements. This ensures that the result is a member of the $\hat{\text{State}}_{RCP}$ and at the same time that it should be larger than each $L \in \mathcal{Y}$.
As a special case of the above, we have that $\biguplus_{\text{RCP}}$ is defined in the following way

$$L_1 \biguplus_{\text{RCP}} L_2 = \{ \hat{\sigma} \mid \hat{\sigma} \in (L_1 \cup L_2), \ \forall \hat{\sigma}' \in (L_1 \cup L_2) : \hat{\sigma} \subseteq_{\text{CP}} \hat{\sigma}' \Rightarrow \hat{\sigma} = \hat{\sigma}' \}$$

Of course now we have to prove that this is in fact a least upper bound operator. And so we first prove that it is an upper bound operator and then that it creates the smallest upper bound.

**Lemma 6.7** $\biguplus_{\text{RCP}}$ is an upper bound operator.

**Proof.** Let $\mathcal{Y}$ be a subset of $\text{State}_{\text{RCP}}$. We prove that the set $\mathcal{L} = \biguplus_{\text{RCP}} \mathcal{Y}$ is an upper bound of $\mathcal{Y}$, that is

$$\forall L' \in \mathcal{Y} : L' \subseteq_{\text{RCP}} \mathcal{L}$$

First of all the Definition 6.6 ensures that $\mathcal{L}$ does not contain any two different mappings that are comparable (i.e. one is smaller than the other), therefore

$$\forall \hat{\sigma}, \hat{\sigma}' \in \mathcal{L} : \hat{\sigma} \subseteq_{\text{CP}} \hat{\sigma}' \Rightarrow \hat{\sigma} = \hat{\sigma}'$$

From the definition of $\biguplus_{\text{RCP}}$ it is also clear that every mapping $\hat{\sigma}' \in L'$ (where $L' \in \mathcal{Y}$), either is incomparable to any other one in $\bigcup \mathcal{Y}$ and thus is a member of $\mathcal{L}$, or is smaller than some other mapping, but then it must be the case that there is a larger one in $\mathcal{L}$. Therefore we can say that

$$\forall L' \in \mathcal{Y} : \forall \hat{\sigma}' \in L' : \exists \hat{\sigma} \in \mathcal{L} : \hat{\sigma}' \subseteq_{\text{CP}} \hat{\sigma}$$

Which is exactly what we need: $\mathcal{L}$ is a member of $\text{State}_{\text{RCP}}$ and it is larger (or equal) than any element of $\mathcal{Y}$. Therefore $\biguplus_{\text{RCP}}$ is an upper bound operator.

**Lemma 6.8** $\biguplus_{\text{RCP}}$ is a least upper bound operator.

**Proof.** Let $\mathcal{Y}$ be a subset of $\text{State}_{\text{RCP}}$ and $\mathcal{L} = \bigcup_{\text{RCP}} \mathcal{Y}$. We prove that the if there exists a set $\mathcal{L}_0 \in \text{State}_{\text{RCP}}$ that is also an upper bound of $\mathcal{Y}$, such that $\mathcal{L}_0 \subseteq_{\text{RCP}} \mathcal{L}$ then $\mathcal{L}_0 = \mathcal{L}$.

Because $\mathcal{L}_0$ is an upper bound of $\mathcal{Y}$ we have:

$$\forall L' \in \mathcal{Y} : \forall \hat{\sigma}' \in L' : \exists \hat{\sigma}_0 \in \mathcal{L}_0 : \hat{\sigma}' \subseteq_{\text{CP}} \hat{\sigma}_0$$

But according to the definition of $\bigcup_{\text{RCP}}$ the set $\mathcal{L}$ is a subset of $\bigcup \mathcal{Y}$, so it must be the case that

$$\forall \hat{\sigma} \in \mathcal{L} : \exists \hat{\sigma}_0 \in \mathcal{L}_0 : \hat{\sigma} \subseteq_{\text{CP}} \hat{\sigma}_0$$
6.2 Relational Constant Propagation

Which means that \( \mathcal{L} \subseteq_{\text{RCP}} \mathcal{L}_0 \). However, we also have \( \mathcal{L}_0 \subseteq_{\text{RCP}} \mathcal{L} \). So from the anti-symmetry of \( \subseteq_{\text{RCP}} \) it follows that \( \mathcal{L}_0 = \mathcal{L} \).

**Theorem 6.9** Partially ordered set \((\widehat{\text{State}}_{\text{RCP}}, \subseteq_{\text{RCP}})\) is a complete lattice.

**Proof.** First of all, recall that for a partially ordered set \( L = (\mathcal{L}, \subseteq) \) the following claims are equivalent

1. \( L \) is a complete lattice.
2. Every subset of \( L \) has a least upper bound.

From 6.8 we know that \( \bigsqcup_{\text{RCP}} \) is a least upper bound operator and so we can find a least upper bound for any subset of \( \widehat{\text{State}}_{\text{RCP}} \). Therefore the second of the above claims is true, and since it is equivalent to the first one we can conclude that \((\widehat{\text{State}}_{\text{RCP}}, \subseteq_{\text{RCP}}, \bigsqcup_{\text{RCP}})\) is a complete lattice.

### 6.2.4 Widening operator

Now that we have defined our lattice for the analysis, it is important to note it does not satisfy the Ascending Chain condition. Recall that a lattice satisfies this condition if and only if all ascending chains eventually stabilise. An ascending chain is a sequence \((l_n)_n, n \in \mathbb{N}\), of elements of a lattice \( L \) such that

\[
    n \leq m \Rightarrow l_n \subseteq l_m
\]

And we say that such a sequence stabilises if and only if

\[
    \exists k \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq k \Rightarrow l_n = l_k
\]

Another important fact to recall is that if \((l_n)_n\) is sequence of elements of \( L \) and \( \phi : L \times L \rightarrow L \) is a total function, then we can construct a new sequence \((l_n^\phi)_n\) in the following way

\[
    l_n^\phi = \begin{cases} 
    l_n & \text{if } n = 0 \\
    l_{n-1}^\phi \phi l_n & \text{if } n > 0 
    \end{cases}
\]

Moreover if the \( \phi \) is an upper bound operator, then the new sequence \((l_n^\phi)_n\) is an ascending chain. This can be proved quite easily, it is enough to show that for all \( n \) we have \( l_n^\phi \subseteq l_{n+1}^\phi \), which can be done by induction.
Now it is clear that, in our case, the lattice does not satisfy the Ascending Chain condition - it still would not be able to handle the example with infinite loop presented before. Therefore, in order to solve this problem, we need to define a widening operator, which will allow us to approximate the least upper bound.

We will say that an operator $\nabla$ is a widening operator if and only if:

1. it is an upper bound operator, and
2. for all ascending chains $(l_n)_n$, the chain $(l_n^\nabla)_n$ eventually stabilises.

First of all, notice that in our case the main problem are variables that might have infinite number of possible values - like in our example with an infinite \textbf{while} loop that increments a counter. One of the possible solutions is to simply limit the number of unique values of each variable, that the analysis will store. And if some variable would have too many values (giving rise to many different mappings stored in the set), we could map the variable to $\top$ (in all the mappings) expressing that the variable can simply have any value.

First of all, let us start with definitions of some simple functions that will be used to create the widening operator.

\textbf{Definition 6.10} The $uvals$ function returns the number of unique values of a variable in the given element of the lattice.

$$uvals(x, L) = |\{\hat{\sigma}(x) \mid \hat{\sigma} \in L\}|$$

Note that we treat $\top$ as one of the possible values. In the following example

$$L = \{[x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 2]\}$$

we would have that $uvals(x, L) = 2$ and $uvals(y, L) = 1$.

We also need a function that would be able to remove the unnecessary mappings in a set and construct one that is in $\text{State}^{\text{RCP}}$ without losing any important information. This is vital because some of the changes that the analysis might perform (e.g. mapping some variables to $\top$), can result in a set that is not an element of the lattice.
6.2 Relational Constant Propagation

Definition 6.11 We define function $mkRCP$ by

$$mkRCP(\mathcal{L}) = \{ \hat{\sigma} \mid \hat{\sigma} \in \mathcal{L}, \forall \hat{\sigma}' \in \mathcal{L} : \hat{\sigma} \sqsubseteq_{cp} \hat{\sigma}' \Rightarrow \hat{\sigma} = \hat{\sigma}'\}$$

It simply filters out elements that are smaller than some other element, leaving only the incomparable ones. This corresponds to transforming an element from $\mathcal{P}(\text{Var}_* \rightarrow \mathbb{Z}^\top)$ to one that is also in $\text{State}_{RCP}$.

Fact 6.12 Observe that from the Definition 6.11 of $mkRCP$ we have

$$\forall \hat{\sigma} \in \mathcal{L} : \exists \hat{\sigma}' \in mkRCP(\mathcal{L}) : \hat{\sigma} \sqsubseteq_{cp} \hat{\sigma}'$$

In other words $mkRCP$ only leaves the mappings that are not smaller than any other ones, so because $\sqsubseteq_{cp}$ is transitive the above property will always hold.

Lemma 6.13 $mkRCP(\mathcal{L})$ creates the largest $\mathcal{L}' \in \text{State}_{RCP}$ such that $\mathcal{L}' \subseteq \mathcal{L}$.

Proof. By way of contradiction let us assume that there exists a set $\mathcal{L}'' \in \text{State}_{RCP}$ such that

$$\mathcal{L}'' \neq \mathcal{L}' \wedge \mathcal{L}' \subseteq \mathcal{L}'' \subseteq \mathcal{L}$$

Then $\mathcal{L}''$ must contain at least one element $\hat{\sigma}''$ (which is also in $\mathcal{L}$) that at the same time is not contained in $\mathcal{L}'$. But from the Fact 6.12 we know that

$$\forall \hat{\sigma} \in \mathcal{L} : \exists \hat{\sigma}' \in \mathcal{L}' : \hat{\sigma} \sqsubseteq_{cp} \hat{\sigma}'$$

So this means that

$$\hat{\sigma}'' \sqsubseteq_{cp} \hat{\sigma}'$$

But as $\hat{\sigma}'$ and $\hat{\sigma}''$ are different and both are members of $\mathcal{L}''$, this contradicts our assumption that $\mathcal{L}'' \in \text{State}_{RCP}$. Therefore $mkRCP(\mathcal{L})$ creates the largest $\mathcal{L}' \in \text{State}_{RCP}$ such that $\mathcal{L}' \subseteq \mathcal{L}$.

Definition 6.14 Finally let us define $\nabla_K$

$$\mathcal{L}_1 \nabla_K \mathcal{L}_2 = \begin{cases} \mathcal{L}_{12} & \text{if } \forall x \in \text{Var}_* : \text{uvals}(x, \mathcal{L}_{12}) \leq K \\ \mathcal{L}'_{12} & \text{otherwise} \end{cases}$$

where $\mathcal{L}_{12} = \mathcal{L}_1 \sqcup_{rcp} \mathcal{L}_2$

$$\mathcal{L}'_{12} = mkRCP(\{ \mathcal{F} \hat{\sigma} \mid \hat{\sigma} \in \mathcal{L}_{12}\})$$

$$\mathcal{F} \hat{\sigma} x = \begin{cases} \top & \text{if } \text{uvals}(x, \mathcal{L}_{12}) > K \\ \hat{\sigma}(x) & \text{if } \text{uvals}(x, \mathcal{L}_{12}) \leq K \end{cases}$$

As you can see if no variable is violating the limit of $K$ values the widening operator is simply $\sqcup_{rcp}$. Otherwise it maps all such variables, in all the mappings to $\top$ and ensures that the resulting set is in $\text{State}_{RCP}$. 
Lemma 6.15 The $\nabla_K$ is an upper bound operator.

Proof. Let:

$$L_3 = L_1 \nabla_K L_2$$

Consider the following cases:

1. If the number of unique values of each of the variables is smaller or equal than $K$ then the result is immediate.

2. Otherwise the variable that has more unique values than $K$ will be mapped to $\top$ in all mappings. Note that, from the definition of $\sqsubseteq_{\text{cp}}$, for any variable $x$ and $\hat{\sigma}$, which contains $x$, we have

$$\hat{\sigma} \sqsubseteq_{\text{cp}} \hat{\sigma}[x \mapsto \top]$$

Now recall that

$$L_3 = \text{mkRCP}(\{F \hat{\sigma} \mid \hat{\sigma} \in (L_1 \sqcup_{\text{rcp}} L_2)\})$$

From the above, the fact that $\sqcup_{\text{rcp}}$ is an upper bound and from the Fact 6.12 we can conclude that:

$$\forall \hat{\sigma} \in L_1 \exists \hat{\sigma}' \in L_3 : \hat{\sigma} \sqsubseteq_{\text{cp}} \hat{\sigma}'$$
$$\forall \hat{\sigma} \in L_2 \exists \hat{\sigma}' \in L_3 : \hat{\sigma} \sqsubseteq_{\text{cp}} \hat{\sigma}'$$

Finally from the definition of $\text{mkRCP}$ we also have that

$$L_3 \in \text{State}_{\text{RCP}}$$

So we can conclude that $L_1 \sqsubseteq_{\text{rcp}} L_3$ and $L_2 \sqsubseteq_{\text{rcp}} L_3$.

Therefore $\nabla_K$ is an upper bound operator.

Theorem 6.16 The $\nabla_K$ is a widening operator.

Proof. A widening operator has to be an upper bound and for all ascending chains $(L_n)_n$ the ascending chain $(L_i^{\nabla_K})_n$ must eventually stabilise. So let us assume that this is not the case and let $(L_i^{\nabla_K})_n$ be an infinite ascending chain that does not stabilise. Now consider two elements of the chain, such that $L_i^{\nabla_K} \sqsubseteq_{\text{rcp}} L_{i+1}^{\nabla_K}$ and $L_i^{\nabla_K} \neq L_{i+1}^{\nabla_K}$. We can observe that at least one of the following must hold:
1. $L_{i+1}^{\triangledown K}$ contains all the mappings that $L_{i}^{\triangledown K}$ has and at least one that is neither smaller or larger than any element of $L_{i}^{\triangledown K}$. This might be possible without increasing the number of unique values of any variable (i.e. by using a new "configuration" of existing values), but as the number of both the variables and existing values is limited, so is the number of such new "configurations". Otherwise it must be the case that at least one of the variables will have more unique values.

2. $L_{i+1}^{\triangledown K}$ contains at least one mapping that is larger (more general) than at least two different elements of $L_{i}^{\triangledown K}$ (which are not in $L_{i+1}^{\triangledown K}$). From the definition of the $\sqsubseteq_{cp}$ ordering we know this might only happen if the state has at least one variable that maps to $\top$.

Now from the definition of $\triangledown K$ the number of unique values of every variable is limited to $K$ and when the limit is about to be exceeded then some more general mappings are created (i.e. having more variables mapped to $\top$). Because the chain is infinite and $K$ is just a finite value, then the second from the above cases will be true infinitely many times. Apart from that, note that if a mapping contains an element mapped to $\top$, then a different, more general one must contain at least two elements mapped to $\top$, and so on. So the number of variables mapped to $\top$ will be increasing. However, as there is only a limited number of variables, eventually a mapping $\hat{\sigma}_M$ will be used/created, which maps all the variables to $\top$. However, this mapping is the largest element of $\text{State}_{cp}$ - it says that every variable can have any value. In other words there exists $m$ such that $L^{\triangledown K}_m = \{\hat{\sigma}_M\}$. But then it must be the case that: $\forall i \in \mathbb{Z} : L^{\triangledown K}_{m+i} = L^{\triangledown K}_m$ which contradicts our assumption. Therefore every infinite ascending chain eventually stabilises.

6.2.5 Transfer functions

The relational CP is operating on the sets of mappings of possible values for the variables used in a program. Intuitively we know that for a given edge with an action, all those mappings should be updated. So here we can take advantage of the fact that those mappings are exactly the same as those in the ordinary CP. Thus for most actions we can simply use the transfer functions from ordinary Constant Propagation analysis. For instance if $x$ is assigned a value of 1, then
we can simply map the transfer function over the whole set of mappings.

\[
f^{\text{RCP}}_{\text{arr}}(\mathcal{L}) = \begin{cases} 
    \text{mkRCP}(\{f^{\text{CP}}_{\text{arr}}(\hat{\sigma}) \mid \hat{\sigma} \in \mathcal{L}, \ B^{\text{CP}}[act][\hat{\sigma}] \neq \text{false}\}) & \text{if } act \text{ is a condition} \\
    \text{mkRCP}(\{f^{\text{CP}}_{\text{arr}}(\hat{\sigma}) \mid \hat{\sigma} \in \mathcal{L}\}) & \text{otherwise}
\end{cases}
\]

where \( \text{arr} = \ell \xrightarrow{act} \ell' \in \rightarrow, \mathcal{L} \in \text{State}_{\text{RCP}} \)

The only difference are the conditions. Here we can simply do much better than in standard CP - we can remove the mappings for which the condition must be false, leaving only those that are either true or unknown (e.g., if some variables used in the expression are mapped to \(\top\)) and still apply the transfer function of independent attribute CP to them. So if we have filtered out the mappings for which a condition is false, we still try to improve the remaining ones. So if we have a set \(\{[x \mapsto \top, y \mapsto 1], [x \mapsto 1, y \mapsto 2]\}\) and have a condition saying \(x = 10\), then we can immediately remove the second mapping, and as for the first one we see that \([x \mapsto 10] \subseteq^{\text{CP}} [x \mapsto \top]\) and there is no other mapping where the condition is true. Thus at the end we have \(\{[x \mapsto 10, y \mapsto 1]\}\).

### 6.2.6 Space complexity

Clearly the relational analysis is much more expensive in terms of space complexity than the independent attribute version. The latter is \(O(nN)\) where \(n\) is the number of variables, and \(N\) is the number of nodes in the program graph. This is quite different in case of relational version. Here we operate on sets of mappings and in the worst case we can have a set that holds \(K^n\) different mappings where \(K\) is the parameter for the widening operator. The reason for this is that the widening operator will allow only \(K\) different values for a given variable (remember that the mappings must not be comparable) so every additional variable will make it possible to have \(K\) times more mappings. So for a program graph with \(n = 2\) and widening operator with \(K = 2\) we can have for instance

\[
\{[x \mapsto 1, y \mapsto 10], [x \mapsto 2, y \mapsto 10], [x \mapsto 1, y \mapsto 20], [x \mapsto 2, y \mapsto 20]\}
\]
At which point adding any new mapping to the set will result in setting one of the variables to $\top$ and the size of the set will not increase. For example

$$ \forall \{ [x \mapsto 1, y \mapsto 10], 
[x \mapsto 2, y \mapsto 10], 
[x \mapsto 1, y \mapsto 20], 
[x \mapsto 2, y \mapsto 20] \} \subseteq \{ [x \mapsto 3, y \mapsto 10] \} = \{ [x \mapsto \top, y \mapsto 10], 
[x \mapsto \top, y \mapsto 20] \} $$

So in the worst case scenario we can have a set of $K^n$ mappings (each having $n$ elements - value for each variable) for each of the $N$ nodes. So the space complexity grows to $O(nK^nN)$.

### 6.2.7 More advanced widening operator

As described above the space complexity of the analysis grows considerably when using relational approach - by a factor of $K^n$ compared to the independent attribute version. As we have no influence on the number of variables, the only way to decrease the memory requirement of our analysis is to keep the $K$ parameter small. However, this can considerably decrease the precision of the analysis. One possible compromise is to notice that not all variables must be analysed with the same precision in order to achieve the desired result. For instance variables used in conditions might be more interesting, because the control flow depends on their values. Here we will explore the idea of making the widening operator be a mapping from variables to integers as the parameter, that is $\bar{K} : \textbf{Var} \rightarrow \mathbb{Z}$. It would indicate the maximum number of unique values separately for each variable.

**Definition 6.17** Having the above in mind we can define the more advanced version of the widening operator in the following way

$$ \mathcal{L}_{1} \triangledown_{\bar{K}} \mathcal{L}_{2} = \begin{cases} 
\mathcal{L}_{12} & \text{if } \forall x \in \textbf{Var}_{+} : \text{uvals}(x, \mathcal{L}_{12}) \leq \bar{K}(x) \\
\mathcal{L}'_{12} & \text{otherwise}
\end{cases} $$

where $\mathcal{L}_{12} = \mathcal{L}_{1} \sqcup \text{rcp} \mathcal{L}_{2}$

$$ \mathcal{L}'_{12} = \text{mkRCP}(\{ \mathcal{F} \hat{\sigma} | \hat{\sigma} \in \mathcal{L}_{12} \}) $$

$$ \mathcal{F} \hat{\sigma} x = \begin{cases} 
\top & \text{if } \text{uvals}(x, \mathcal{L}_{12}) > \bar{K}(x) \\
\hat{\sigma}(x) & \text{if } \text{uvals}(x, \mathcal{L}_{12}) \leq \bar{K}(x)
\end{cases} $$
The only difference with the previous widening operator is the use of $\bar{K}(x)$ instead of $K$.

**Lemma 6.18** The $\nabla_{\bar{K}}$ is an upper bound operator.

**Proof.** The proof is almost identical to the one for $\nabla_K$ (6.15). The only difference is that the variables will be mapped to $\top$ only when their limit (and not a global one) is exceeded.

**Theorem 6.19** The $\nabla_{\bar{K}}$ is a widening operator.

**Proof.** And again the proof is almost the same as in the case of the previous widening operator, see 6.16. The only thing that needs additional attention is that we are concerned with the chain $\mathcal{L}^{\bar{K}}$ and that every variable has a separate limit. However, the reasoning does not change as the limits are still simply integers and allow only a finite number of unique values for every variable.

### 6.3 Comparison

In order to emphasize the limitations of the independent attribute Constant Propagation and at the same time present how the relational approach works and what are its advantages, we will have a look at an example. The simple program below decrements $x$ until it is 0 and at the same time increments $y$. The end result is that $x$ will be 0 and $y$ will have the value that $x$ had at the beginning of the loop.

```plaintext
x := 2;
y := 0;
while x > 0 do (
y := y + 1;
x := x - 1
)
```

Now the ordinary CP will not be able to provide any interesting results. The fact that both of the variables are changed in the while loop will result in the analysis assuming that both of them are $\top$. The output of our program
is presented in Figure 6.1. So we do not really learn anything at all about the program! Fortunately we can also use the relational approach and depending on the parameter $\bar{K}$ we will be able to get the precise values of any of the variables. Compare the Figure 6.2 that was generated with $\bar{K} = [x \mapsto 3, y \mapsto 3]$. The limit of 3 values is the smallest one that gives the precise answer to this problem (have a look at node 3). However, it is interesting to note that no matter how high the limit for $y$ is, it will not provide any interesting result unless the $x$ limit is high enough. This is exactly because without knowing the values of $x$ the analysis does not know how many times the body of the loop will get executed.
Figure 6.2: Results of the relational CP with $\tilde{K} = [x \mapsto 3, \ y \mapsto 3]$. 
Chapter 7

Interesting examples

7.1 Dekker’s algorithm

7.1.1 Introduction

Dekker’s algorithm is the first known solution to the mutual exclusion problem. The algorithm is presented in the Table 7.1. Note that the algorithm assumes that turn is initialised either to 0 or 1. As our implementation does initialise all the variables to 0 by default, we can use that and skip any initialisation part\(^1\) of the algorithm.

Thanks to the fact that the program graphs are so easy to compose, we can create one that will express both of the processes executing concurrently. And then we can actually analyse it using the already defined analyses.

\(^1\)This means that the processes are numbered as process 0 and process 1.
flag0 := 1;
while flag1 == 1 do
    if turn == 1
        then (flag0 := 0;
            while turn == 1 do skip;
            flag0 := 1)
    else skip;
// critical section
critical := 0;
// leave critical section
flag0 := 0;
turn := 1

flag1 := 1;
while flag0 == 1 do
    if turn == 0
        then (flag1 := 0;
            while turn == 0 do skip;
            flag1 := 1)
    else skip;
// critical section
critical := 1;
// leave critical section
flag1 := 0;
turn := 0

Table 7.1: Dekker’s algorithm

As the algorithm should ensure mutual exclusion it is natural to try to check whether it actually does. First of all it is necessary to decide how it can be checked using the program graph. In our algorithm the critical section is simply represented by assignment to a variable critical. Therefore we should be looking for a node with at least two outgoing edges that assign to this variable (i.e. have associated assignment actions). If the results of the analysis indicate that this node is reachable then we might have a problem, otherwise the algorithm is safe. We will use the query mechanism implemented in the application to find and highlight the node that satisfies a query. In this case we use the following one

$$\exists \ell, \ell' : \text{mod}(\text{Node}, \text{critical}, \ell) \& \text{mod}(\text{Node}, \text{critical}, \ell') \& \ell \neq \ell'$$

Where Node is the node we are searching for, \(\ell\) and \(\ell'\) are some neighbouring nodes, mod is a predicate indicating that a variable is changed on the given edge. More information about the query mechanism is available in the section 8.2.2 in the next chapter.

Generating all the interleavings of those two processes will obviously create the node in question. Now the way to check whether the algorithm is actually secure is to show that it is simply not reachable. And this in turn might be possible with the Constant Propagation analysis, if it is able to determine the
Figure 7.1: Dekker’s algorithm analysed by the independent attribute CP.
values of different variables and allow us to remove the edges that will never be executed, i.e. when the condition associated with the edge is always false with the provided results of the analysis.

The result of the independent attribute CP analysis is clearly showing that there is no possibility of both processes entering the critical section at the same time. The graph is presented in the Figure 7.1. It is interesting to note that the analysis is not able to determine the values of variables at the end of the program, but is powerful enough to conclude that the node, we are interested in, is not reachable.

### 7.2 Peterson’s algorithm

#### 7.2.1 Introduction

Peterson’s algorithm is another classic algorithm for mutual exclusion. It also allows two processes to access the critical section without conflict using only shared memory for communication. The algorithm in the \texttt{WHILE} language is presented in the Table 7.2. This algorithm does not really need any initialisation part as the \texttt{turn} variable is assigned by the processes themselves. Therefore we will leave it at the default value 0, and index the processes as 1 and 2 - this way they have the same chance of entering the critical section first (which was not the case in the previous example).

The algorithm is a little bit more concise than the Dekker’s one. However, it is also somewhat more difficult to analyse. The \texttt{turn} variable is modified by both

| flag1 := 1; | flag2 := 1; |
| turn := 2; | turn := 1; |
| while flag2 == 1 and turn == 2 do skip; | while flag1 == 1 and turn == 1 do skip; |
| critical := 1; | critical := 2; |
| flag1 := 0 | flag2 := 0 |

Table 7.2: Peterson’s algorithm
7.2 Peterson’s algorithm

| turn := 2; | turn := 1; |
| flag1 := 1; | flag2 := 1; |
| (while flag2 == 1 and turn == 2 do skip); | (while flag1 == 1 and turn == 1 do skip); |
| critical := 1; | critical := 2; |
| flag1 := 0 | flag2 := 0 |

Table 7.3: Modified Peterson’s algorithm.

processes at the beginning and the conditions (and thus the algorithm) rely that certain combinations of values of turn and flag1/2 variables are not possible. And because of that the analysis is not capable of determining that the critical section will not be entered by both of the processes at the same time, see Figure 7.2. As expected the analysis is having trouble with the turn variable, because it sets it to ⊤ and so it cannot be certain that the conditional edges going to the undesired node are always false. Fortunately we also have the relational version of the analysis. Using it we can get the result presented in the Figure 7.3. This time the node 33 is not present on the graph at all! It was removed as it is simply not reachable. And this is possible because the analysis is able to determine the precise values of the turn and flag1/2 variables. So using the relational Constant Propagation, we have confirmed that the algorithm does ensure mutual exclusion.

7.2.2 Modified Peterson’s algorithm

So now we can go on and experiment a little bit with the algorithm, for instance we can easily check what happens if we change the order of the first two assignments in both processes - Table 7.3. After analysing this algorithm using the relational CP, it turns out that both processes can actually enter the critical section at the same time - Figure 7.4. So the order of this assignment is crucial for the mutual exclusion property. And with the wrong one there exists an interleaving that gives rise to a situation where both processes are in the critical section at the same time. One such interleaving is presented in the Table 7.4.

Finally we can try to fix the algorithm using the atomic statement. Let us take the ”broken” version but this time make both assignments atomically, so that there is no interleaving between the assignments. The Table 7.5 presents the updated algorithm. And this time the algorithm again works as expected, and
Interesting examples

Process 1

\[
\text{turn} := 2; \quad \text{flag2} := 1; \quad (\text{while flag1 == 1 and turn == 1 do skip}); \\
\text{CRITICAL SECTION}
\]

\[
\text{flag1} := 1; \quad (\text{while flag2 == 1 and turn == 2 do skip}); \\
\text{CRITICAL SECTION}
\]

Table 7.4: Interleaving where both processes enter the critical section

\[
\text{atomic(turn := 2; flag1 := 1);} \quad \text{atomic(turn := 1; flag2 := 1);} \\
(\text{while flag2 == 1 and turn == 2 do skip);} \quad (\text{while flag1 == 1 and turn == 1 do skip}); \\
\text{critical := 1;} \quad \text{critical := 2;} \\
\text{flag1 := 0} \quad \text{flag2 := 0}
\]

Table 7.5: Peterson's algorithm with atomic statements.

there is no possibility for both of the processes to get into the critical section at the very same time, see Figure 7.5.
Figure 7.2: Peterson’s algorithm analysed by the independent attribute CP.
Figure 7.3: Peterson’s algorithm analysed by the relational CP.
Figure 7.4: Modified Peterson's algorithm analysed by the relational CP.
Figure 7.5: Peterson’s algorithm with atomic statements analysed by the relational CP.
Chapter 8

Implementation

8.1 Introduction

The thesis also includes an implementation of the ideas presented in the previous chapters. It is written mostly in Haskell with some small parts written in HTML, CSS and JavaScript, which are used by the user interface. Haskell is a purely functional programming language with the call-by-need semantics (often called a lazy programming language). It was chosen for this project because of its high level of abstraction, powerful static type system and rich, well supported libraries. The implementation is a web application and so requires a web browser to use it. It allows the user to write down a number of WHILE programs and generate and analyse the corresponding program graph. The results are presented in form of an interactive HTML page with PNG image of a graph and some JavaScript code for easy and convenient inspection of the analysis results. The whole implementation is just around 2500 lines of code, including the parsers, graph generation, user interface, solving algorithms and most importantly all the analyses.
There are basically two groups of analyses that are implemented in the application. There are Reaching Definitions and Live Variables analyses that are expressed in the form of ALFP, then solved by the Succinct Solver and finally interpreted and presented to the user. And there is also an internal solving algorithm (i.e. the worklist algorithm). The main motivation for it was of course the Constant Propagation analysis, as it cannot be translated into ALFP (neither the independent attribute and relational version). Moreover versions of RD and LV using it has also been added to the list of available analyses.

The project is fully cabalised\(^1\). This makes it quite easy to install, as all the dependencies on Haskell packages should be resolved automatically. The exact procedure to install and run the software is provided in the Appendix A.

### 8.2 Using the application

#### 8.2.1 Overview

As already mentioned the implementation is simply a web application and thus can be access and used using a web browser. It does use some JavaScript, so it should be enabled for the interface to work correctly. The interface, presented in Figure 8.1, is quite simple and allows to write down any number of WHILE programs, that will be automatically combined into a single program graph. Moreover it is possible to write a query that will result in highlighting the nodes satisfying a given constraints on the resulting graph. In case of the relational Constant Propagation, it is also possible to specify the parameter for the widening operator.

Finally it should be mentioned that for both versions of Constant Propagation, there are additional algorithms that process the resulting graph. Their main goal is to remove edges that will never be taken (based on the results of the analysis), such as edges with conditions that are false. Afterwards the nodes that are not reachable anymore, are also removed. This is very important in our considerations in the Chapter 7 on Dekker’s and Peterson’s algorithms.

\(^1\)Cabal is a system for building and packaging Haskell libraries and programs - http://www.haskell.org/cabal/
8.2 Using the application

Figure 8.1: The main interface of the implementation.
8.2.2 Query mechanism

The queries in current implementation can only be used for finding nodes that satisfy certain properties, relating to incoming or outgoing edges. Because of that, the label of the node, that one is searching for, has been predefined as Node. Moreover all the additional labels that you might introduce using the $\exists$ quantifier (in the application itself written simply as E), only refer to the neighbouring nodes. So for example the query

$$E\ n:\ use(\text{Node}, \text{var}, n)$$

means that every node that has an outgoing edge using the variable var will be highlighted. Similarly

$$E\ n:\ mod(n, \text{var}, \text{Node})$$

means that every node that has an incoming edge modifying variable var will be highlighted. Apart from that, if one introduces more than one neighbouring node

$$E\ n_1, n_2:\ mod(\text{Node}, \text{var}, n_1) \& mod(\text{Node}, \text{var}, n_2)$$

it does not imply that the $n_1$ and $n_2$ are different. So if the query should actually find nodes that have two different outgoing edges modifying var one must add $n_1 \neq n_2$ to the above.

8.2.3 Widening operator parameters

The widening operator used for the relational Constant Propagation analysis can be defined only when this analysis has been chosen from the menu. For simplicity, it is treated as a global parameter for all the variables, and then there is a simple way to override it for some specified variables. So the functionality of the advanced operator where every variable can have a different limit is available. At the same time the interface is a little bit easier to manage than in the case when the user would have to define it separately for each of the variables.
8.2.4 Succinct Solver integration

The current implementation uses the Succinct Solver V2.0. It is implemented in SML programming language and thus SML NJ must be installed if the analyses using the solver are to be used. As the application needs to find and run sml or sml.bat executable, it is important to have an environment variable SMLNJ_HOME defined and pointing to the main directory of SML NJ installation. More information about installing SML NJ is available on its homepage [http://www.smlnj.org/](http://www.smlnj.org/). Note that the Succinct Solver V2.0 might not work when using the most recent version of SML NJ. It was tested that everything works as expected using version 110.59.

8.3 Project structure

8.3.1 Common

There are a few files with common functionality that is used by most of the other modules. Obviously one of them is Types.hs where all the main data types, type class definitions and some of their instances are defined. Apart from that there is Common.hs that has some common functions potentially useful for various modules. Moreover, there is a Pretty.hs that defines the type class for pretty printing and the instances for all the necessary data types. Finally there is also a very simple Error.hs with basic definitions of possible errors used by other modules.

8.3.2 Parsing

There are a total of three parsers used in the project. Obviously there is one for the WHILE language. Apart from that there is one for the query mechanism and yet another one that parses the output of the Succinct Solver.
Implementation

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WhileParser.hs</td>
<td>parser for the WHILE language</td>
</tr>
<tr>
<td>SolverParser.hs</td>
<td>parser for the Succinct Solver results</td>
</tr>
<tr>
<td>QueryParser.hs</td>
<td>parser for the queries</td>
</tr>
</tbody>
</table>

All of the parsers use the Parsec monadic parser combinator library. One of the main advantages of using it, is that all the parsers are written directly in Haskell, without any additional yacc-like languages. Moreover, Parsec has also a module that provides a way to easily tokenize the input, which is used in the parsers for WHILE and for queries. Thus no additional lexers are required.

8.3.3 Graphs

For representing graphs, the FGL (Functional Graph Library\(^2\) package has been used. The concepts behind it were introduced in \(^2\) and currently it is probably the most commonly graph library for Haskell. Thanks to that, there are quite a few other libraries/packages that accept graphs as defined by the FGL, making it much easier to incorporate various features in the application (this might be also important in any future extensions). One such example, that is actually part of the implementation, is Graphviz package\(^3\) that provides binding to a library of the same name. It does accept an input in the form of an FGL graph and uses the original Graphviz library\(^4\) for the graph generation. In order to achieve the ability to show and hide the results of the analysis by clicking the nodes of the graph, two possibilities has been considered. Initially SVG format seemed to offer the best performance and possibility to make it somewhat interactive using JavaScript. However, it seems that currently the SVG support in most browsers is not satisfactory. Therefore it was decided to use PNG file format and additionally generate a client-side image map, that is then appended to the HTML page. It contains calls to JavaScript functions that make it possible to conveniently inspect the results of the analysis at different nodes.

\(^2\)http://hackage.haskell.org/package/fgl
\(^3\)http://hackage.haskell.org/package/graphviz
\(^4\)www.graphviz.org
8.3 Project structure

8.3.4 "Internal" analyses

The Monotone Framework is defined using a type class, which every analysis must be an instance of. This makes the worklist algorithm quite simple and easy to understand. Moreover, the analyses themselves can also be expressed quite concisely. Of course, the RD and LV are quite simple, whereas CP is a little bit more demanding, as it requires the ability to evaluate arithmetic and boolean expressions\(^5\). Finally, the relational analysis is the most complex. It uses a bit more complex lattice and must maintain the right structure of used sets (in the implementation represented as lists). Apart from that, it uses the independent attribute version for all the mappings inside those sets.

- `MFSolver.hs`: solving algorithm
- `RD.hs`: Reaching Definitions
- `LV.hs`: Live Variables
- `CPCommon.hs`: some common functions used by both CP and RCP
- `CP.hs`: independent attribute Constant Propagation
- `RCP.hs`: relational Constant Propagation

8.3.5 ALFP Analyses/Succinct Solver

The ALFP analyses are defined in quite a different way than the ones using the internal solver. Here, the main goal is to create a number of clauses that can be used as input for the solver. Some clauses must be generated for each assignment, or read statement, some can be defined globally. Apart from that, for each of the analyses, one must generate some predicates expressing the structure of the graph, such as `Edge` or `Label`, that the analyses simply expect to be defined. Obviously, it uses the already mentioned pretty printer for ALFP. Moreover, a special wrapper around the solver has been created, that takes care of creating temporary file writing the clauses to it and calling the solver itself. Finally, the result is parsed and interpreted by a function that is provided by the given analysis. This is necessary as different analyses have different results; in this case, we have RD that has two terms to be interpreted (the variable identifier and the label) and LV that only uses variable identifiers.

\(^5\)Boolean expressions are only important for the small improvement regarding the conditional edges.
<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALFPTypes.hs</td>
<td>data types for ALFP</td>
</tr>
<tr>
<td>ALFPCommon.hs</td>
<td>common functions</td>
</tr>
<tr>
<td>SolverAnalysis.hs</td>
<td>general interface for the analyses</td>
</tr>
<tr>
<td>SolverRD.hs</td>
<td>Reaching Definitions in ALFP</td>
</tr>
<tr>
<td>SolverLV.hs</td>
<td>Live Variables in ALFP</td>
</tr>
<tr>
<td>SolverParser.hs</td>
<td>parser for Succinct Solver V2.0 output</td>
</tr>
<tr>
<td>SolverIntegration.hs</td>
<td>wrapper around the solver</td>
</tr>
</tbody>
</table>

### 8.3.6 Web interface

As already mentioned the program works as a web application. From the server-side it has been implemented using the Happstack\(^6\) (Haskell application server stack). This allowed to use Haskell for most of the interface and user interaction code with all the usual benefits of this language. Moreover it provided a nice integration with the rest of the code-base. On the client-side JavaScript was used to simplify some of the interface and make it more usable. This includes dynamically adding or removing textareas for user input, showing/hiding or highlighting the results of the analysis. In order to simplify this task (and to make the result more appealing) one of the most popular JavaScript libraries, jQuery\(^7\), has been used. Of course all of the HTML positioning, layout, colors has been defined using CSS (Cascading Style Sheets).

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HtmlPages.hs</td>
<td>HTML generation</td>
</tr>
<tr>
<td>Server.hs</td>
<td>the Happstack-based implementation of the interface</td>
</tr>
</tbody>
</table>

\(^6\)[http://happstack.com/](http://happstack.com/)

\(^7\)[http://jquery.com/](http://jquery.com/)
Chapter 9

Conclusions

9.1 Summary

The thesis explored the possibility of using program graphs as the basis for Data Flow Analysis. First the program graphs were introduced (Chapter 1) and formally defined (Chapter 2). As an example programming language, we used WHILE and defined its semantics, how to create a program graph from WHILE and finally proved that whatever WHILE program does, it will have the same effect in the program graph (Chapter 3). Moreover, a number of analyses have been redefined to work on program graphs, including Reaching Definitions, Live Variables and Constant Propagation. What is more, the thesis also presented how to express the first two in in Alternation-free Least Fixed Point Logic so that the result can be obtained using the Succinct Solver (Chapters 4 and 5). Apart from that, a relational version of Constant Propagation analysis has been defined in Chapter 6. It is quite a bit more complex and expensive than the other analyses, but it proved especially useful when analysing concurrently executing processes, where both of them might modify the same variable. In such a situation independent attribute analysis is simply not precise enough.
The thesis also presented one of the main advantages of program graphs, that is the ability to compose them. This allows to simulate a shared memory environment with multiple processes executing concurrently. With the Constant Propagation analyses it made it possible to analyse and experiment with some classical mutual exclusion algorithms as presented in the Chapter 7.

Finally the thesis also includes an implementation of the above ideas. It is a web application written in Haskell, that allows to write a number of WHILE programs, create graphs for them, merge them into one graph and finally analyse it. It has a interactive interface and utilises Graphviz library for generating graphs. It also has a query mechanism that is capable of finding nodes in such graphs based on properties such as outgoing edges that modify some variable. Finally, it implements all of the analyses considered in the thesis.

9.2 Future work

The thesis does open a few possibilities for future work. First of all, only slightly modified WHILE programming language has been used, which is, in fact, quite simple. It could be interesting to try and extend it with more features. One of them might be to introduce procedures, and then try to define the interprocedural analyses. Apart from that, as the program graph approach can be used to analyse concurrently executing processes, it could be interesting to add some synchronisation and/or communication primitives. This would probably require some more changes, not only to the semantics of program graphs, but also to the way they are created and merged. Such an extension would allow to analyse much more advanced algorithms and programs.

What is more, the implementation of analyses themselves could be improved. For instance, it might be worth extending the capabilities of Constant Propagation with respect to edges with conditions. Apart from that, in case of the relational version, it is possible to specify the parameter for the widening operator, where higher limits improve the precision of the analysis, but are also more expensive. The current implementation requires the user to specify the parameter. It could be exciting to explore the possibility of approximating the parameter automatically, maybe by using some other analysis or simply examining the abstract syntax tree.
9.2 Future work

Apart from that, the query mechanism is quite simple. Making it more powerful could allow searching for multiple nodes, edges, or even paths. Moreover, there is some room for improvement in the way the results are presented to the user. For instance when merging a number of program graphs into one, the resulting graph can be quite huge. One could consider, for instance, implementing a way to generate only parts of the graph, or make the graph itself more dynamic, with the functionality of collapsing parts of it.

To conclude, there are many interesting possibilities of extending the presented idea, creating analyses that are more precise and analysing more complex programs.
Howto install and run the application

A.1 Requirements

The project requires GHC (Glasgow Haskell Compiler\(^1\)). It can be downloaded as a single package, but currently it is recommended to obtain it as a part of the Haskell Platform available at http://hackage.haskell.org/platform/, which additionally includes a number of important and popular libraries.

Note for users of GHC 6.12.2. There is a bug that has been introduced in the GHC 6.12.2, which incidentally changed the behaviour of one of the basic functions and thus makes Happstack unusable. For more details see the bug report at http://hackage.haskell.org/trac/ghc/ticket/3808. The new version of Happstack (i.e. $\geq 0.5$) includes a workaround for this bug and works fine.

The application uses the Haskell bindings for Graphviz library to generate the

\(^1\)http://www.haskell.org/ghc/
How to install and run the application

graphs, but as these are just bindings, they do require the actual library to be present on the system. Therefore it is necessary to install it before using the application. It is available at http://www.graphviz.org/.

Apart from that, the application requires a number of Haskell libraries. They are all available on Hackage\(^2\), which is a central collection of released Haskell packages As the next section shows, hey should be quite easy to install.

A.2 Installing and running

The application is cabalised, i.e. uses the cabal build system for easy dependency resolution and building. More details about cabal are available on its website at http://www.haskell.org/cabal/. Thanks to cabal, the installation should require simply unpacking the archive, entering the project directory and typing:

\[
> \text{cabal install}
\]

Then cabal should resolve the dependencies and install all the required Haskell packages automatically. Please remember that it might be necessary to run

\[
> \text{cabal update}
\]

to get the most recent list of available packages. Alternatively, if one does not want to install the application itself, it is possible to install the dependencies by hand and build/use the application from its directory. Running

\[
> \text{cabal configure}
\]

will list all the missing libraries. After installing them using cabal install command, it should be enough to run

\(^2\)http://hackage.haskell.org/
> cabal build

to build the application.

After having it installed, it should be possible to simply call the application from the command line\(^3\). It has only two options

\[\text{\verb+webanalyse -h+}\]
\[\text{\verb+webanalyse [-d <directory>] [-p <port>]}\]

Without any argument the application will run inside the current directory and listen on address \texttt{localhost:8080}. However, as the application requires the directory to contain the necessary JavaScript, CSS files, and the Succinct Solver, it should be started inside the "html" subdirectory (located inside the project directory), or this directory should be specified using the "-d" switch. So either

> cd html
> webanalyse

or alternatively

> webanalyse -d html

would run the application.

\(^3\text{On Windows the application will be called }\texttt{webanalyse.exe}\)
How to install and run the application
Bibliography


