Generating Threads for Non-strict Functional Programming Languages

by

Christiana Virginia Toutet

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Copyright 1998 Christiana Virginia Toutet. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author .................................................................

Department of Electrical Engineering and Computer Science

May 8, 1998

Certified by .................................................................

Arvind
Professor
Thesis Supervisor

Accepted by .................................................................

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Generating Threads for Non-strict Functional Programming Languages

by

Christiana Virginia Toutet

Submitted to the Department of Electrical Engineering and Computer Science on May 8, 1998, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In an implicitly parallel language only a partial order on instructions may be specified. However, standard processors are adapted to exploiting sequential instructions. The task of ordering instructions is not straightforward for a non-strict language, where functions may return values before all their arguments are available, and data structures may be defined before all their components are. Since the compiler may not be able to determine a total ordering, partitioning corresponds to breaking a program into sequential parts, threads, whose relative ordering is dictated by run-time.

Unlike previous work, our algorithm is completely syntax directed and integrates both basic blocks, as well as general blocks. It does not depend on an intermediate dataflow representation, or a dependence graph. Therefore, conditionals, function calls, and recursive function definitions are treated systematically. Furthermore, by introducing a new representation, \textit{predicated sets}, information about every possible flow of control is encapsulated such that the general block algorithm takes all dependence possibilities into consideration simultaneously, rather than requiring a separate analysis for each possibility. As a result of predicated sets, a new fixed point termination condition on an infinite domain is also introduced based on the use of the analysis results.

Thesis Supervisor: Arvind
Title: Professor
Acknowledgments

I am very grateful to my thesis advisor Professor Arvind for his guidance and support, as well as to Jan-Willem Maessen for his continuous help.

Of course I would also like to thank Bine, George, Gert, and Pooh. No comment needed. They know who they are, and they know best all they have done for me.
To the sweat and tears of Villennes Plage,
Contents

1 Introduction 10
  1.1 The Problem .............................................. 11
  1.2 Related Work .............................................. 13
    1.2.1 Demand/tolerance algorithm .......................... 13
    1.2.2 Dependence, demand and tolerance sets ............... 13
    1.2.3 The algorithm ........................................ 14
    1.2.4 An example .......................................... 14
    1.2.5 Analyzing recursion using paths ..................... 15

2 SubId: Syntax and Semantics 18
  2.1 Abstract Syntax .......................................... 19
  2.2 Operational Semantics .................................... 20
    2.2.1 Contexts ............................................ 20
    2.2.2 Rewrite Rules ....................................... 20

3 Predicated Sets 22
  3.1 Predicated Set Grammar ................................... 23
  3.2 Predicated Set Constructor Operations .................... 23
    3.2.1 Axioms ............................................. 23
    3.2.2 Canonical form for predicated sets .................. 25
  3.3 Structure of the Predicated Set Domain .................. 26
  3.4 Operations On Predicated Sets ............................ 27
    3.4.1 Properties of the \( \cap_p \) operator ................ 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties of the ( \cup_p ) operator</td>
<td>28</td>
</tr>
<tr>
<td>Properties of ( S_p )</td>
<td>28</td>
</tr>
<tr>
<td>Properties of ( R_p )</td>
<td>29</td>
</tr>
<tr>
<td>Preserving the canonical form</td>
<td>29</td>
</tr>
<tr>
<td>4 Abstracting Paths</td>
<td>33</td>
</tr>
<tr>
<td>4.1 Formalizing Paths</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1 A formal path abstraction</td>
<td>33</td>
</tr>
<tr>
<td>4.1.2 Path environments</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Abstraction Rules</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Preliminary definition</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Abstraction rules</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Discussion and Correctness</td>
<td>38</td>
</tr>
<tr>
<td>4.3.1 Property of paths</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2 Simple expressions</td>
<td>39</td>
</tr>
<tr>
<td>4.3.3 Primitive functions</td>
<td>40</td>
</tr>
<tr>
<td>4.3.4 Conditionals</td>
<td>40</td>
</tr>
<tr>
<td>4.3.5 User defined function calls</td>
<td>41</td>
</tr>
<tr>
<td>4.3.6 Statements within branches of a conditional</td>
<td>43</td>
</tr>
<tr>
<td>4.3.7 Function definitions</td>
<td>44</td>
</tr>
<tr>
<td>5 Dependence, Demand and Tolerance Labels</td>
<td>48</td>
</tr>
<tr>
<td>5.1 Definitions</td>
<td>48</td>
</tr>
<tr>
<td>5.2 Correctness of Labels for Basic Blocks</td>
<td>49</td>
</tr>
<tr>
<td>5.3 Observations for General Blocks</td>
<td>53</td>
</tr>
<tr>
<td>6 From Source Code to Partitioned Code</td>
<td>55</td>
</tr>
<tr>
<td>6.1 Partitioning</td>
<td>55</td>
</tr>
<tr>
<td>6.2 Partitioned Code Grammar</td>
<td>56</td>
</tr>
<tr>
<td>6.3 Algorithm</td>
<td>56</td>
</tr>
<tr>
<td>6.3.1 Compute labels necessary for partitioning</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Implicitly parallel languages such as Id[7] and pH[6](a parallel dialect Haskell[4]) are closely linked to the evolution of dataflow architectures since custom-built dataflow machines like Monsoon[9] offer an ideal execution model allowing for large instruction-level parallelism. In the dataflow execution model, execution of an operation is enabled by availability of the required operand values. The completion of one operation makes the resulting values available to the elements of the program whose execution depends on them. This differs from sequential machines where a program counter register holds the address of the next instruction to be executed. Despite the advantages of dataflow architecture for parallelism, the Von Neumann architecture has remained prevalent. Therefore, considerable effort has been directed towards implementing languages such as Id on standard processors.

Standard processors are adapted to exploiting ordered instructions since the most widespread languages, like C, are sequential. Therefore, efficiently executing a program written in a language such as Id on a sequential or parallel machine built out of Von Neumann processors requires ordering instructions. The task of ordering instructions is not straightforward because these languages exploit non-strictness to provide implicit parallelism. In non-strict functional languages functions may return values before all their arguments are defined, and data-structures may be defined before all their components are. Therefore, one can, for example, feed outputs of a function back into its inputs. The simple paradigm of waiting for all inputs to be defined and
then calling the function can no longer be applied, as it might lead to deadlock.

1.1 The Problem

Description

A total compile time ordering may be impossible to obtain since a “correct” instruction order may vary according to the inputs. Intuitively, an ordering is “incorrect” if it leads to deadlock while there exists another ordering which does not. Consider the following example written in Id:

```idl
f(a, b) = { x = a + 2;
y = b * 2;
in
(x, y)}
```

This snippet of Id code represents a function $f$ taking inputs $a$ and $b$. The code delimited by braces corresponds to a block which contains one or more bindings, the keyword “In,” and a body which is an expression. The bindings associate identifiers with the value of expressions. In our example $x$ is bound to the value of $a + 2$ and $y$ is bound to the value $b * 2$. The block as a whole represents a value, the value of the body.

In our example, no order for execution of the statements contained in the block is specified, so in which order should the compiler choose to compute $x$ and $y$? In this case, an ordering cannot be determined at compile time while preserving the non-strict semantics of the language. Consider the following two contexts:

```idl
{ ... 
(x, y) = f(1, x) 
...}
```

```idl
{ ... 
(x, y) = f(y, 1) 
...}
```
In the first context, $y$ cannot be computed until $x$ is computed, since the value of $x$ is fed back into the input of $f$. Computing $y$ before $x$ leads to deadlock, whereas computing $x$ before $y$ leads to the value $(3, 6)$. In the second context, $x$ cannot be computed until $y$ is computed, since $y$ is fed back into the input of $f$. Computing $x$ before $y$ leads to deadlock, whereas computing $y$ before $x$ leads to the result $(4, 2)$.

Objective

The compiler may not be able to determine a total ordering at compile time. Therefore, the goal of partitioning is to sequentialize parts of a program by inferring possible dependencies. The relative ordering of these sequentialized parts, threads, is dictated by run-time. Each thread must satisfy the following:

1. Instructions within a thread can be ordered at compile time such that the ordering is correct for all contexts in which the procedure may be invoked.

2. Once the dependencies are satisfied for a thread’s execution, all statements can be computed without interruption.

3. Values computed in an executing thread are not visible to other threads before the thread completes.

To minimize the run-time overhead incurred in ordering the threads, partitions should be as large as possible. Furthermore, large partitions allow for machine-level optimizations to allocate resources, such as registers, more efficiently. While finding the largest threads possible is an NP-complete problem[2], the aim is to find a good approximation at compile time.

In the rest of this chapter, we give a brief overview of the two partitioning algorithms on which we based our approach. In chapter 2, we give the syntax and semantics of the language we will be working with. In chapter 3, we introduce a representation, predicated sets, for the abstract interpretation presented in chapter
4. In chapter 5, the results of abstract interpretation, paths, are used to compute dependence, demand, and tolerance labels. In chapter 6, partitioned code is produced according to the information contained in these labels. Chapter 7 concludes.

1.2 Related Work

The partitioning problem was first explored by Ken Traub in [8].

1.2.1 Demand/tolerance algorithm

The algorithm we propose produces partitions equivalent to those produced by the demand/tolerance algorithm[3] with respect to basic blocks—blocks whose operator nodes are all primitive operators. However, our algorithm differs from this algorithm for general blocks in the way it handles conditionals, procedure calls, and recursive procedures. Furthermore, the demand/tolerance algorithm is not syntax directed. It handles programs represented as dataflow graphs:

**Definition 1 (Dataflow Graphs)** Given a block b, the dataflow graph G(b) corresponding to b is a directed graph. The Nodes N(b) in the graph consist of input nodes—corresponding to the free variables of the block, output nodes—corresponding to the values returned by the block, and operator nodes—corresponding to the various operators specified in the block. The edges E(b) of this graph correspond to the flow of values in the block.

We therefore only give an overview of this algorithm for basic blocks.

1.2.2 Dependence, demand and tolerance sets

The algorithm uses demand, dependence, and tolerance sets to determine which nodes can be put in the same partition.

**Definition 2 (Dependence)** The dependence set of a node in the basic block BB is the set of input nodes on which it depends.
Dep(i) = \{i\} if i is an input node.
Dep(n) = \bigcup_{(m,n) \in BB} Dep(m)

**Definition 3 (Demand)** The demand set of a node in a basic block BB is the set of output nodes which depend on it.
Dem(n) = \{o\} if o is an output node.
Dem(n) = \bigcup_{(n,m) \in BB} Dem(m)

**Definition 4 (Tolerance)** An output o tolerates a node n if and only if an edge (n, o) can be added to BB without affecting input/output connectivity. The tolerance set of a node n is the set of outputs that tolerate n.
Tol(o) = Dem(o) if o is an output. Tol(n) = \bigcap_{(m,n) \in BB} Dem(m)

### 1.2.3 The algorithm

Given a basic block b.

1. Compute demand and tolerance sets for each node of b
2. Do
   (a) Merge nodes with the same demand set into a single partition. Recompute tolerance sets from the reduced graph.
   (b) Merge nodes with the same tolerance set into a single partition. Recompute demand sets from reduced graph,

until the partitions do not change.

### 1.2.4 An example

\[
\begin{align*}
f(a,b) &= \{ \\
    n_1 &= a + 1 \\
    n_2 &= b \times 2 \\
    n_3 &= a + b \\
    r_1 &= n_1 + n_2
\end{align*}
\]
Starting by merging all nodes with the same tolerance would yield one partition in one step. However, if all nodes with the same demand sets are merged first, we would obtain three partitions on the first step: nodes $a, b$, nodes $n_1, n_2, r_1$, and nodes $n_3, r_2$. Tolerance sets do not change as a result of the merging. Therefore, on the next step all nodes would be combined into the same partition.

1.2.5 Analyzing recursion using paths

The demand/tolerance algorithm is not able to handle recursion satisfactorily. Algorithms involving abstract interpretation, such as strictness analysis, are more adapted to handling recursion. The algorithm proposed by Satyan Coorg[3] uses an abstract interpretation framework similar to the one we present in this thesis. His algorithm approaches the partitioning problem for general blocks, including conditionals, user defined functions, as well as recursive functions, in two steps. First, a path analysis infers the possible dependencies of a program. Then, the results of the abstract interpretation step are used to produce a partitioning with the demand/tolerance approach.

Path analysis

Definition 5 (Paths) A path is either a set of variables or $\perp_p$. In the Path domain $\perp_p$ is the least element. The paths for a function $f$ with $N$ arguments and returning $M$
results is a set of $M$-tuples of paths. An individual path $[u_1, \ldots, u_k]$ denotes that if the function $f$ takes that path, the value corresponding to the path depends on $u_1, \ldots, u_k$ and is always defined if $u_1, \ldots, u_k$.

Example for a conditional:

$$h \ x \ y = \{ r = \{ \text{if } y \text{ then } x \text{ else } 1 \} \text{ in } r\}$$

$f$ has a set of paths $\{[y, x], [y]\}$.

The algorithm

Given a block $b$.

1. Select a node in $b$ that calls some function $f$. Disconnect all other nodes that are not primitive operators. Disconnecting a node involves introducing a new output node corresponding to each of the function’s arguments and a new input node corresponding to each of the function’s results.

2. Let $f$ be an $M$-result function and have $P$ paths. For each $i \geq j \geq P$ construct block $b_j$ as follows:

   (a) Let the $j^{th}$ path be $(p_{j_1}, \ldots, p_{j_M})$

   (b) For each $1 \geq k \geq M$ do

      i. Let $O$ be the nodes of $b$ connected to the $k^{th}$ output of $f$

      ii. If $p_{j_k} = \bot_p$, connect each node in $O$ to a new input node called $\bot$.

      iii. Otherwise, introduce a new “dummy” node $(op)$ with inputs connected to the arguments of $p$ in $p_{j_k}$. Connect $op$ with each node in $O$.

   (c) For $1 \geq j \geq P$, delete all nodes in $b_j$ that are transitively connected to a $\bot$ node.
(d) Rename all input and output nodes so that there are no conflicts of names between blocks.

(e) Apply the demand/tolerance basic block partitioning of nodes in $b$.

(f) Go to step 1 if changes to the partitioning of $b$ occur.

**Limitations**

Coorg’s approach separates the case of partitioning basic blocks from general blocks. A more integrated approach which no longer refers to dataflow and is completely syntax directed would be more desirable. In this algorithm, partitioning general blocks also entails constructing a basic block for each possible flow of control through the program. Then, the demand/tolerance algorithm is run on each of the derived basic blocks. This thesis extends Coorg’s algorithm by adopting a similar approach to partitioning which is more integrated and avoids Demand/Tolerance iterating over each possible basic block.
Chapter 2

SubId: Syntax and Semantics

Before describing the partitioning algorithm, we present the language, SubId, we will be basing our analysis on. SubId determines the scope of our analysis. SubId is a first order, functional language with simple data types and a formal operational semantics. The language is composed of basic blocks (blocks containing only primitive functions), conditionals, and first order, recursive functions. SubId’s functions can have any number of inputs and outputs. However, SubId does not include recursive let blocks, higher order functions, or data-structures and they are not covered in our analysis. When partitioning a general block containing a function call, we assume that we have previously analyzed the function.

Although our analysis does not include letrec blocks, the let block we are partitioning may occur within a letrec block.

\[
f(x_1, \ldots, x_n) =
\]

letrec

\[
y_1, \ldots, y_k = \text{LetBlock}
\]
in

\[
y_1, \ldots, y_k
\]

Therefore, the partitions must be correct for any possible context the let block might occur in. For the moment, we will define a partitioning as correct if it does not intro-
duce a deadlock. Since partitions for let blocks must be correct under any possible feedback from an output to an input, partitioning a let block is not trivial; a simple topological sort does not suffice.

2.1 Abstract Syntax

To make the abstract interpretation step clearer, we assume that the statements in the let blocks have been topologically sorted according to the their variables dependencies prior to our analysis. The grammar of SubId is given below:

\[
\begin{align*}
\text{TopLevel} & ::= F(x_1, \ldots, x_k) = E \\
E & ::= SE \\
& \quad \mid \{S \text{ in } x_1, \ldots, x_k\} \\
& \quad \mid PF_k(x_1, \ldots, x_k) \\
& \quad \mid F_k(x_1, \ldots, x_k) \\
& \quad \mid \text{Cond}(SE, E_t, E_f) \\
S & ::= x_1, \ldots, x_k = E \\
& \quad \mid S; S \\
SE & ::= x \mid V \\
V & ::= CN_0 \\
PF_1 & ::= \text{negate} \mid \text{not} \mid \ldots \mid Prj_1 \mid Prj_2 \mid \ldots \\
CN_0 & ::= \text{Number} \mid \text{Boolean}
\end{align*}
\]

where:
represents the set of  

Primitive functions

Expressions

Simple Expressions

Values

Constants

are instances of Variables

2.2 Operational Semantics

The operational semantics for SubId is given in terms of a set of rewrite rules and the contexts in which these rules can be applied. These are similar to the rules of λs, an operational semantics for pH [1]. (We assume that all variables are α renamed.)

2.2.1 Contexts

\[ C\[ \] := [ ] \]
\[ \mid \{SC\[ \] in x_1,\ldots,x_k\} \]
\[ \mid \{S in x_1,\ldots,\[\],x_k\} \]
\[ \mid PF_k(x_1,\ldots,\[\],\ldots,x_k) \]
\[ \mid F_k(x_1,\ldots,\[\],\ldots,x_k) \]
\[ \mid Cond([\],E_t,E_f) \]
\[ SC\[ \] := x_1,\ldots,x_k = C\[ \] \]
\[ \mid SC\[ \]; S \]

2.2.2 Rewrite Rules

\[ \beta \text{ rule} \]
\[ F(x_1,\ldots,x_k) \rightarrow \{S in r_1,\ldots,r_n\}[x_1/t_1,\ldots,x_k/t_k] \]
where \( F(t_1,\ldots,t_k) = \{S in r_1,\ldots,r_n\} \)
and \([x_1/t_1, \ldots, x_k/t_k]\) means substitute \(x_i\) for every occurrence of \(t_i\).

**\(\delta\) rule**

\[PF(v_1, \ldots, v_k) \rightarrow pf(v_1, \ldots, v_k)\]

where \(pf(v_1, \ldots, v_k)\) is the result of applying \(PF\) to \(v_1, \ldots, v_k\).

**Cond rule**

\[Cond(True, E_t, E_f) \rightarrow E_t\]

\[Cond(False, E_t, E_f) \rightarrow E_f\]

**Flat rule**

\[S'',\]

\[\{\ y_1, \ldots, y_k = \{ S \]

\[\quad \text{in} \ x_1, \ldots, x_k\};\]

\[S'\]

\[\text{in} \ r_1, \ldots, r_n\}\]

\[\rightarrow\]

\[S'';\]

\[\{\ S;\]

\[y_1 = x_1;\]

\[\ldots\]

\[y_k = x_k;\]

\[S'\]

\[\text{in} \ r_1, \ldots, r_n\}\]
Chapter 3

Predicated Sets

Our analysis is built within an abstract interpretation framework in order to handle recursion using fixed point iteration. The analysis is composed of two parts: first, we compute the dependence related information (similar to dependence, demand and tolerance sets [3]) and encapsulate it in a representation. Subsequently, we use the information for partitioning. In this chapter, we focus on the underlying representation used for expressing dependence information.

The same two step approach appears in [3] which uses sets to encapsulate dependence information. However, information contained in sets alone is not enough to produce correct partitions. Consider, for example, the case of a conditional in which we may obtain different variable dependencies according to whether the predicate is true or false. More precisely, suppose two expressions, $E_1$ and $E_2$, have the same possible dependence sets, \{\{a, c\}, \{b, c\}\}, and share the same predicate $p$. Knowing that both expressions have the same dependencies is not enough for us to deduce that both expressions will depend on the same variables under the same circumstances. When $p$ is true, $E_1$ could depend on \{a, c\} while $E_2$ could depend on \{b, c\}. However, this information is needed to determine whether $E_1$ and $E_2$ can safely be put into the same partition.

Thus, our choice of representation, while closely linked to sets, is slightly different. It keeps track of the context in which each set occurs. In this way, continuing with the example above, we are able to tell whether $E_1$ and $E_2$ both have sets \{a, b\} or
\{b, c\} when \(p\) is true. We will refer to our representation as predicated sets.

In this chapter, we describe the predicated set grammar, algebra, and domain structure.

### 3.1 Predicated Set Grammar

A predicated set is constructed over a domain \(V\). Predicated sets are constructed using the \(o\), \(\land\), and + operations.

\[
P = \text{vars} \\
| P + P \\
| \text{pred} o P \\
| \bot \\
\text{pred} = b_x | \overline{b_x}
\]

\(x \in V, \text{vars} \subseteq V\).

The motivation for this representation is to associate a context with each possibility we need to consider. For the purposes of our analysis, \(V\) corresponds to the domain of variables. The notation \(b_x\) means that \(x\) is a predicate variable whose value is true, and \(\overline{b}_x\) means that \(x\) is a predicate variable whose value is false. The + operation separates sets occurring in different contexts, and the \(o\) operation attaches a context to a predicated set. Thus, \(b_x o \{x_1, x_2\} + \overline{b}_x o \{x_3, x_4\}\), means when predicate variable \(x\) is true we consider the set \(\{x_1, x_2\}\), and when predicate \(x\) is false we consider the set \(\{x_3, x_4\}\).

### 3.2 Predicated Set Constructor Operations

#### 3.2.1 Axioms

The \(o\) operation

\[
pred_1 o \ pred_2 o P = pred_2 o \ pred_1 o P
\]
The \( o \) operation associates a context with a predicated set. Specifying the same \( pred \) twice gives no additional information as to which context the predicated set occurs in. Also, the order of the \( pred \)'s within the context does not matter.

**The + operation**

\[
\begin{align*}
P + P & = P \\
P_1 + P_2 & = P_2 + P_1 \\
P_1 + (P_2 + P_3) & = (P_1 + P_2) + P_3 \\
vars_1 + vars_2 & = vars_1 \cup vars_2 \\
vars + \bot & = \bot \\
pred o P_1 + pred o P_2 & = pred o (P_1 + P_2) \\
b_x o P_1 + \overline{b_x} o P_1 & = P_1 \\
P_1 + b_x o \overline{b_x} o P_2 & = P_1 \\
pred o \{\} & = \{\}
\end{align*}
\]

The + operation is used to separate predicated sets which share a different context in order to express mutually exclusive possibilities. So \( b_x o \{x_1\} + \overline{b_x} o \{x_2\} \), for example, means that there are two cases to consider, that is if \( x \) is true or false. Therefore, the intuition behind a rule like \( b_x o \{x_1\} + \overline{b_x} o \{x_1\} = \{x_1\} \) is that no matter what the value of the predicate \( x \), we are considering the set \( \{x_1\} \). Note that \( b_x o \{\} \neq \overline{b_x} o \{\} \neq \{\} \); therefore, the following rule is false: \( P + \{\} = P \).
3.2.2 Canonical form for predicated sets

In order to work with predicated sets in the clearest way, we adopt a canonical form which corresponds to expanding the predicated sets to a sum of terms, \( context_1 \ o \ vars_1 + \ldots + context_n \ o \ vars_n \), with mutually exclusive contexts, meaning \( (context_i \ o \ context_j \ o \ A) + B = B, \forall i \neq j \).

Let us show that any predicated set conforming to the grammar can be rewritten as \( \bot \), \( vars \), or a sum of terms, \( term_1 + \ldots + term_n \), with the following form:

1. \( \bot \)

2. \( \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \bot \), where \( j > 0 \).

3. \( \{x_1,\ldots,x_i\} \), where \( i \geq 0 \).

4. \( \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \{x_1,\ldots,x_k\} \), where \( j > 0 \) and \( k \geq 0 \).

Induction proof on the grammar:

Base Case: \( \bot \), \( vars \) are of the right form.

Induction Step: Suppose \( P_1 \) and \( P_2 \) are of the right form, let us show that \( P_1 + P_2 \), as well as \( \text{pred} \ o \ P_1 \) are of the right form. Let \( P_1 = term_1 + \ldots + term_n \) and \( P_2 = term'_1 + \ldots + term'_m \).

1. \( P_1 + P_2 \): Trivial, since \( P_1 + P_2 = term_1 + \ldots + term_n + term'_1 + \ldots + term'_m \) and by the hypothesis each term is of the right form.

2. \( \text{pred} \ o \ P_1 \)
   
   (a) \( P_1 \) is \( \bot \): \( \text{pred} \ o \ P_1 = \text{pred} \ o \ \bot \)

   (b) \( P_1 \) is \( \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \bot \): \( \text{pred} \ o \ P_1 = \text{pred} \ o \ \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \ \bot \)

   (c) \( P_1 \) is \( \{x_1,\ldots,x_i\} \): \( \text{pred} \ o \ P_1 = \text{pred} \ o \ \{x_1,\ldots,x_i\} \)

   (d) \( P_1 \) is \( \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \{x_1,\ldots,x_k\} \):

      \( \text{pred} \ o \ P_1 = \text{pred} \ o \ \text{pred}_1 \ o \ldots \ o \text{pred}_j \ o \ \{x_1,\ldots,x_k\} \)
Once our paths are expressed as above, obtaining a canonical form for paths is straightforward using rules for the $\circ$ and $+$ operators which collapse predicated sets with the same contexts, as well as expand predicated sets into their different contexts. For example, $\{\} + b_x \circ \{x\}$ can be rewritten as:

$$\overline{b_x} \circ \{\} + b_x \circ \{x\} = \overline{b_x} \circ \{\} + b_x \circ (\{\} + \{x\})$$

$$= \overline{b_x} \circ \{\} + b_x \circ \{x\}$$

### 3.3 Structure of the Predicated Set Domain

In this section we introduce an ordering on the predicated sets. This involves an element $\bot$ which we have ignored so far, although we introduced it in the grammar and algebra. $\bot$ is the bottom of the predicated set domain. We have the following rule for $\bot$: $\bot + \text{vars} = \bot$. However, if $\bot$ is the bottom element then the following rules must hold:

$$\text{pred} \circ \bot \subseteq \text{pred} \circ P$$

$$P + \bot \subseteq P + P'$$

We will use these constraints to determine an order on the domain.

Let us first look at what the latter rules tell us in the case where $P$ and $P'$ are simply variables, call them $x$ and $y$. Then

$$\text{pred} \circ \bot \subseteq \text{pred} \circ \{x\}$$

$$\{x\} + \bot = \bot \subseteq \{x\} + \{y\} = \{x, y\}$$

If we introduce contexts, then the same constraints yield:

$$b_x \circ \{x\} + \overline{b_x} \circ \{y\} + P' \supseteq b_x \circ \{x\} + \overline{b_x} \circ \{y\} + \bot$$

$$= (b_x \circ \{x\} + \overline{b_x} \circ \{y\}) + (b_x \circ \bot + \overline{b_x} \circ \bot)$$

$$= b_x \circ (\{x\} + \bot) + \overline{b_x} \circ (\{y\} + \bot)$$

$$= \bot$$
"Less than" means "unsafe approximation to". Therefore, \( \bot \) will represent an unsafe approximation to any other predicated set. Our constraints give us little indication as to the ordering of predicated sets. For example, how does \( b_x \circ \bot \) compare to \( \bot \)? Therefore, we will introduce the following rule: Let \( P = \text{context}_1 \circ s_1 + \ldots + \text{context}_n \circ s_n \) and \( P' = \text{context}_1 \circ s_1 + \ldots + \text{context}_m \circ s_m \) be two predicated sets in canonical form and \( s = \text{vars} | \bot \) then:

\[ P \subseteq P' \text{ iff } \forall i \in [1, n], s_i \subseteq s'_i \text{ and } n < m. \]

Continuing with the previous example mentioned, \( \bot = b_x \circ \bot + b_x \circ \bot \) so \( b_x \circ \bot \subseteq \bot \).

### 3.4 Operations On Predicated Sets

We also need some operators to manipulate predicated sets:

\[ Op = P \cup_p P \mid P \cap_p P \mid S_p P \mid R_p P \]

Union and intersection will be used later to obtain the demand, tolerance and dependence information necessary for identifying which variables can safely be put in the same partition. \( S_p \) and \( R_p \) will be used to express our abstraction rules.

#### 3.4.1 Properties of the \( \cap_p \) operator

Taking the intersection of two predicated sets \( P_1 \) and \( P_2 \) is analogous to taking the intersection of two sets sharing the same context. For example, \( (b_x \circ \{x_1, x_2\} + \overline{b_x} \circ \{x_1, x_3\}) \cap (b_x \circ \{x_1\} + \overline{b_x} \circ \{x_3\}) \) corresponds to taking the intersection of \( \{x_1, x_2\} \) with \( \{x_1\} \) when \( x \) is true, and the intersection of \( \{x_1, x_3\} \) with \( \{x_3\} \) when \( x \) is false which yields in our notation \( b_x \circ \{x_1\} + \overline{b_x} \circ \{x_3\} \).

\[
\begin{align*}
P \cap_p P &= P \\
P_1 \cap_p P_2 &= P_2 \cap_p P_1 \\
(P_1 \cap_p P_2) \cap_p P_3 &= P_1 \cap_p (P_2 \cap_p P_3) \\
(pred_1 \circ P_1) \cap_p P_2 &= pred_1 \circ (P_1 \cap_p P_2)
\end{align*}
\]
\[ P_1 \cap_P (P_2 + P_3) = (P_1 \cap_P P_2) + (P_1 \cap_P P_3) \]

\[ \text{vars} \cap_P \text{vars'} = \text{vars} \cap \text{vars'} \]

\[ \bot \cap_P \text{vars} = \text{vars} \]

\[ P_1 \cap_P (P_2 \cup_P P_3) = (P_1 \cap_P P_2) \cup_P (P_1 \cap_P P_3) \]

3.4.2 Properties of the \( \cup_P \) operator

Taking the union of two predicated sets \( P_1 \) and \( P_2 \) corresponds to taking the union of sets occurring within the same context. For example, \((b_x o \{x_1, x_2\} + \overline{b_x} o \{x_3\}) \cup_P (b_x o \{x_1\} + \overline{b_x} o \{x_4\})\) corresponds to taking the union of \( \{x_1, x_2\} \) with \( \{x_3\} \) when \( x \) is true, and the union of \( \{x_3\} \) with \( \{x_4\} \) when \( x \) is false which yields in our notation \( b_x o \{x_1, x_2, x_3\} + \overline{b_x} o \{x_3, x_4\} \).

\[ P_1 \cup_P P_2 = P_1 + P_2 \]

\[ P_1 \cup_P (P_2 \cap_P P_3) = P_1 \cup_P P_2 \cap_P P_1 \cup_P P_3 \]

Union behaves just like the + operation since the union of two predicated sets \( P_1 \) and \( P_2 \) produces a predicated set whose terms are the ones which combine the sets of \( P_1 \) and \( P_2 \). The only difference with the + operation is the additional distributive property of \( \cup_P \) over \( \cap_P \).

3.4.3 Properties of \( S_p \)

Let \( S_p = \{x_1 = p_1, \ldots, x_n = p_n\} \) be a substitution. Then

\[ S_p \bot = \bot \]

\[ S_p \{x_i\} = p_i \]

\[ S_p \{x'\} = \{x'\} \text{ if } \forall i, x' \neq x_i \]

\[ S_p \{\} = \{\} \]
Sp pred o p = pred o Sp p
Sp p1 + p2 = Sp p1 + Sp p2

From properties above we derive the following:
Sp \{y_1, \ldots, y_n\} = Sp \{y_1\} + \ldots + Sp \{y_m\}

3.4.4 Properties of R_p

Let R_p = [x'_1/x_1, \ldots, x'_n/x_n] be a renaming.

p[x'/x] cases on p

\begin{align*}
\bot &= \bot \\
\{x\} &= x' \\
\{y\} &= y \\
(p_1 + p_2)[x'/x] &= p_1[x'/x] + p_2[x'/x] \\
pred o p &= pred[x'/x] o p[x'/x] \\
b_x[x'/x] &= b_x \\
b_y[x'/x] &= b_y \\
\overline{b}_x[x'/x'] &= \overline{b}_x \\
\overline{b}_y[x'/x'] &= \overline{b}_y
\end{align*}

From the properties above we derive the following:
\{y_1, \ldots, y_m\} = \{y_1\}[x/x'] + \ldots + \{y_m\}[x/x']

3.4.5 Preserving the canonical form

The result of applying \cap_p, \cup_p, S_p, and R_p to predicated sets yields a predicated set which can be rewritten as \bot, \{\}, or a sum of terms, \text{term}_1 + \ldots + \text{term}_n, with the following form:

1. \bot

2. \text{pred}_1 o \ldots o \text{pred}_j o \bot, where \ j > 0.
3. \( \{x_1, \ldots, x_i\} \), where \( i \geq 0 \)

4. \( \text{pred}_1 o \ldots o \text{pred}_j o \{x_1, \ldots, x_k\} \), where \( j > 0 \) and \( k \geq 0 \)

**Proof.** We know that any predicated sets obtained from the grammar have the required form, and that the operators \( o \) and \( + \) preserve this form. Since \( \cap_p \) and \( \cup_p \) are defined in terms of \( o \) and \( + \), they must also preserve the form.

Suppose \( P_1, P_2 \) are of the right form, let us show that \( P_1 \cup_p P_2 \) as well as \( P_1 \cap_p P_2 \) are of the right form. Let \( P_1 = \text{term}_1 + \ldots + \text{term}_n \) and \( P_2 = \text{term}'_1 + \ldots + \text{term}'_m \)

1. \( P_1 \cup_p P_2 \): \( P_1 \cup_p P_2 = P_1 + P_2 \) which is of the required form.

2. \( P_1 \cap_p P_2 \):

\[
P_1 \cap_p P_2 = \text{term}_1 \cap_p \text{term}'_1 + \ldots + \text{term}_n \cap_p \text{term}'_m
\]
\[
+ \ldots
\]
\[
+ \text{term}_n \cap_p \text{term}'_1 + \ldots + \text{term}_n \cap_p \text{term}'_m
\]

Therefore, to show that \( P_1 + P_2 \) has the right form we show that each \( \text{term}_v \cap_p \text{term}'_w \) for \( 0 < v < n+1 \) and \( 0 < w < n+1 \) has the right form. Since intersection is commutative, we need not exhaustively examine every case.

(a) \( \text{term}_v \) is \( \perp \):

i. \( \text{term}'_w \) is \( \perp \): \( \text{term}_v \cap_p \text{term}'_w = \perp \)

ii. \( \text{term}'_w \) is \( \text{pred}_1 o \ldots o \text{pred}_j o \perp \):

\[
\text{term}_v \cap_p \text{term}'_w = \text{pred}_1 o \ldots o \text{pred}_j o (\perp \cap_p \perp)
\]
\[
= \perp
\]

iii. \( \text{term}'_w \) is \( \{x_1, \ldots, x_i\} \):

\[
\text{term}_v \cap_p \text{term}'_w = \{x_1, \ldots, x_i\} \cap_p \perp
\]
\[
= \perp
\]

iv. \( \text{term}'_w \) is \( \text{pred}_1 o \ldots o \text{pred}_j o \{x_1, \ldots, x_k\} \):

\[
\text{term}_v \cap_p \text{term}'_w = \text{pred}_1 o \ldots o \text{pred}_j o (\{x_1, \ldots, x_k\} \cap_p \perp)
\]
\[
= \text{pred}_1 o \ldots o \text{pred}_j o \perp
\]

28
(b) $\text{term}_w$ is $\text{pred}_1 o \ldots o \text{pred}_j o \bot$: similar to the previous case since, $\text{term}_w \cap_p \text{term'}_w = \text{pred}_1 o \ldots o \text{pred}_j o (\bot \cap_p \text{term'}_w)$ and we know the form of $\bot \cap_p \text{term'}_w$.

(c) $\text{term}_w$ is $\{x_1, \ldots, x_i\}$:
   
   i. $\text{term'}_w$ is $\{x'_1, \ldots, x'_k\}$:
      
      $\text{term}_w \cap_p \text{term'}_w = \{x_1, \ldots, x_i\} \cap_p \{x'_1, \ldots, x'_k\}$
   
   ii. $\text{term'}_w$ is $\text{pred}_1 o \ldots o \text{pred}_j o \{x'_1, \ldots, x'_k\}$ then
      
      $\text{term}_w \cap_p \text{term'}_w = \{x_1, \ldots, x_i\} \cap_p \text{pred}_1 o \ldots o \text{pred}_j o \{x'_1, \ldots, x'_k\}$
      
      $= \text{pred}_1 o \ldots o \text{pred}_j o (\{x'_1, \ldots, x'_k\} \cap_p \{x_1, \ldots, x_i\})$

(d) $\text{term}_w$ is $\text{pred}_1 o \ldots o \text{pred}_j o \{x_1, \ldots, x_k\}$:
   
   $\text{term'}_w$ is $\text{pred'}_1 o \ldots o \text{pred'}_j o \{x'_1, \ldots, x'_k\}$
   
   $\text{term}_w \cap_p \text{term'}_w = \text{pred}_1 o \ldots o \text{pred}_j o \{x_1, \ldots, x_k\} \cap_p$
   
   $\text{pred'}_1 o \ldots o \text{pred'}_j o \{x'_1, \ldots, x'_k\}$
   
   $= \text{pred}_1 o \ldots o \text{pred}_j o \text{pred'}_1 o \ldots o \text{pred'}_j o (\{x_1, \ldots, x_k\} \cap_p$
   
   $\{x'_1, \ldots, x'_k\})$

3. Let us show $S_p = \{x_1 = p_1, \ldots, x_n = p_n\}$ also preserves the canonical form.

   Base Case: $S_p \{x_i\} = p_i$, is of the right form if $p_i$ is of the right form.

   $S_p \{\}\ = \{\}$, $S_p \bot = \bot$, $S_p \{y\} = y$ where $\forall i, y \neq x_i$ are of the right form.

   $S_p \text{ vars} = S_p \{y_1, \ldots, y_m\} = S_p \{y_1\} + \ldots + S_p \{y_m\}$ is of the right form

   because from above $S_p \{y_1\}$ is of the right form.

   Induction Step: Let $S_p p$ be the right form. Let us show that $S_p (p_1 + p_2)$ and $S_p (\text{pred} o p)$ are of the right form.

   (a) $S_p (p_1 + p_2) = S_p p_1 + S_p p_2$ is of the right form since $S_p p_1$ and $S_p p_2$ are of the right form (hypothesis).

   (b) $S_p \text{ pred} o p = \text{pred} o S_p p$, since $S_p p$ is of the right form, we know $\text{pred} o S_p p$ is of the right form from the canonical form of predicated sets.
4. $R_p = [x_1/x_1, \ldots, x'_n/x_n]$ also preserves the canonical form. Trivial.
Chapter 4

Abstracting Paths

In order to determine which variables can safely be put in the same partition, we first find the path of every variable in the general block we are analyzing. The path of a variable \( x \) can be thought of as the set of variables needed to produce the value of \( x \). However, according to the run-time values, we may obtain different possible paths. Consider, for example, the case of a conditional in which we have two paths, according to whether the predicate is true or false. Therefore, we use predicated sets to represent the path of a variable since predicated sets allow us to represent different possibilities and associate these with the context they occur in. In this chapter we describe the formal path abstraction as well as the precise rules for deriving paths directly from SubId syntax.

4.1 Formalizing Paths

4.1.1 A formal path abstraction

Path of a variable

The path of a variable \( x \) is the predicated set consisting of all the variables needed to compute \( x \) and the context in which they are needed. To make our abstraction rules more readable, the path of \( x \) also contains itself for each possible context.

For example:
<table>
<thead>
<tr>
<th>Variable</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w )</td>
<td>{ w }</td>
</tr>
<tr>
<td>( y )</td>
<td>( b_w \circ { y, z, w } )</td>
</tr>
<tr>
<td>( z )</td>
<td>{ z }</td>
</tr>
<tr>
<td>( x )</td>
<td>( b_w \circ { w, y, z, x } + b_w \circ { w, x } )</td>
</tr>
</tbody>
</table>

\[
x = \begin{cases} 
\text{if } w \\
\{ y = 3 \times z \\
in y \} \text{ then } \{ y = 3 \times z \\
in y \} \\
\text{else } 1 \end{cases}
\]

where \( w \) and \( z \) are inputs.

In order to compute \( y \) we need the value of \( z \) and \( w \). To compute \( x \) we need \( w \). If \( w \) is true we also need \( y \) and \( z \). If \( w \) is false we do not need any additional variables. Therefore, we obtain the following paths:

**Paths of an expression**

To find the path of an expression, predicated sets are not expressive enough. In SubId, an expression may produce a tuple of values. Therefore, the path scheme of an expression is a tuple of paths, where each path corresponds to an output of the given expression. Therefore we introduce Paths:

\[
Paths := Path, Paths | Path.
\]

**Path scheme of a procedure**

The function scheme contains the formal parameters of the function as well as the path for each output. We will use the following formalism to express the path scheme of a function:

\[
\rho := \lambda x_1 \ldots x_n . p_1 , \ldots , p_k
\]

For example, the path scheme of \( f \) where:

\[
f(x, y) = \{ r = x \ast y \ast 2; \}
\]
\[ r' = 1 \]
in \( r, r' \}

is \( \lambda xy.\{x, y, r\}, \{r'\} \)

In order to work with function path schemes we provide the following definitions:

**Definition 6 (Path Variable)**

\[
PV(P) = \text{case on } P
\]
\[
\triangleright \text{pred o } P' = PV(\text{pred}) \cup PV(P')
\]
\[
\triangleright P' + P'' = PV(P') \cup PV(P'')
\]
\[
\triangleright \text{vars} = \text{vars}
\]
\[
\triangleright b_x = \{x\}
\]
\[
\triangleright \overline{b}_x = \{x\}
\]

**Definition 7 (Input Variables)** Let \( \rho = \lambda x_1 \ldots x_n. p_1, \ldots, p_k \)

\( \text{InpV}(\rho) = \{x_1, \ldots, x_n\} \)

**Definition 8 (Internal Variables)** Let \( \rho = \lambda x_1 \ldots x_n. p_1, \ldots, p_k \)

\( \text{IntV}(\rho) = PV(p_1) \cup \ldots \cup PV(p_n) - \{x_1, \ldots, x_n\} \)

### 4.1.2 Path environments

**Variable path environment**

A variable path environment, \( B \), maps variables to their respective paths:

\[ B : \text{var} \rightarrow \text{path} \]

An example variable path environment is \( \{x : \{x\}, y : b_x o \{x, z\} + \overline{b}_x o \{x\}, z : \{z\}\} \).

**Function path environment**

A function path environment, \( f \), maps functions to their respective path schemes:

\[ f : \text{function} \rightarrow \rho. \]
An example function path environment is \( \{ F : \lambda xy.(\{x, y\}; \{\}) \} \).

4.2 Abstraction Rules

The abstraction rules will give us:

1. a variable path environment containing a path for every variable in the block which we will use to partition a given block

2. a function path environment containing a path for every function analyzed so far which is used for inter-block analysis.

4.2.1 Preliminary definition

In the abstraction rules for expressions, we will need the following definition.

**Definition 9 (Bound Variables)** Let the bound variables of an expression, \( BV(E) \), be the variables that occur on the left hand side of the statements comprised in \( E \) (where \( E \) is an expression in \( \text{SubId} \)). We define \( BV(E) \) as follows:

\[
BV(E) = \text{case of } E \\
\{x_1, \ldots, x_n = E'; S \text{ in } y_1, \ldots, y_k\} = \{x_1, \ldots, x_n\} \cup_p BV(\{S \text{ in } y_1, \ldots, y_k\}) \\
\text{else} \\
= \{\}
\]

where \( E \) is an expression of \( \text{SubId} \).

4.2.2 Abstraction rules

Signature of the abstraction functions \( A \) and \( P \)

\[
A : \text{Toplevel} \rightarrow f \rightarrow (B, \text{path}, f) \\
P : E \rightarrow B \rightarrow f \rightarrow (B, \text{path}, f) \\
B : \text{path var} \rightarrow \text{path} \\
f : \text{function name} \rightarrow \text{path scheme}
\]
f initially maps everything to ⊥.

Rules

\[ P[[C_{\emptyset}]]Bf = (B, \text{empty}, f) \]
\[ P[[x]]Bf = (B, B(x), f) \]
\[ P[[y_1, \ldots, y_n]]Bf = (B, (B(y_1), \ldots, B(y_n)), f) \]
\[ P[[\{x_1, \ldots, x_n = E; S \text{ in } y_1, \ldots, y_n\}]]Bf = \text{let } (B', (p_1, \ldots, p_n), f) = P[[E]]Bf \]
\[ B'' = B' + \{x_j = p_1 + x_1, \ldots, x_n = p_n + x_n\} \]
\[ \text{in } P[[\{S \text{ in } y_1, \ldots, y_n\}]]B''f \]
\[ P[[PF(a_1, \ldots, a_k)]]Bf = \lambda x_1 \ldots x_k. \text{paths} = f(F) \]
\[ t_1, \ldots, t_m = \text{IntV}(\lambda x_1 \ldots x_k. \text{path}) \]
\[ \text{paths}' = \text{paths}[t'_1/t_1, \ldots, t'_m/t_m, x_1, \ldots, x_k] \]
\[ S = \{a_1 = B(a_1), \ldots, a_k = B(a_k)\}, t'_1 = \{\}, \ldots, t'_m = \{\} \]
\[ \text{in } (B, S \ \text{paths}', f) \]
\[ P[[\text{Cond}(x, E_t, E_f)]]Bf = \text{let } (B', (p_1, \ldots, p_k), f) = P[[E_t]]Bf \]
\[ r_1, \ldots, r_n = \text{BV}(E_t) \]
\[ (B''', (p'_1, \ldots, p'_k), f) = P[[E_f]]Bf \]
\[ r'_1, \ldots, r''_m = \text{BV}(E_f) \]
\[ B''' = B + \{x = B'(r_1) + x\}, \ldots, r_n = B'(r_n) + x\} \]
\[ + \{x = B''(r'_1) + x\}, \ldots, r''_m = B''(r''_m) + x\} \]
\[ \text{in } (B''', (B_x o (p_1 + \{x\}) + B_x o (p'_1 + \{x\})), f) \]
\[ \ldots, B_x o (p_k + \{x\}) + B_x o (p'_k + \{x\}), f) \]
\[ A[[F(x_1, \ldots, x_n) = \{S \text{ in } y_1, \ldots, y_k\}]]f = \text{let } (B', (p_1, \ldots, p_k), f) = \]
\[ P[[\{S \text{ in } y_1, \ldots, y_k\}]](x_1 = x_1, \ldots, x_n = x_n)f \]
\[ f' = f + \{F : \lambda x_1 \ldots x_n. p_1, \ldots, p_k\} \]
\[ \text{in } A[[F(x_1, \ldots, x_n) = \{S \text{ in } y_1, \ldots, y_k\}]]f' \]

with the following termination condition:
Let \(\lambda x_1 \ldots x_n. p^k_1, \ldots p^k_k = f^k(h)\); we terminate when \(t(p^k_i) = t(p^{k-1}_i)\) for every \(h\) and every \(p_i\).  \(t\) strips the predicate information from canonicalized terms (terms where
all elements of sums have pairwise disjoint contexts) as follows:

\[ t(p_1 + p_2) = t(p_1) \cup t(p_2) \]
\[ t(\text{pred } o \ p) = t(p) \]
\[ t(\text{vars}) = \{\text{vars}\} \]
\[ t(\bot) = \{\bot\} \]

### 4.3 Discussion and Correctness

In this section we discuss the rules and prove the path environment \( B \) we obtain is correct by showing that if:

1. for some context, \( C \), corresponding to a choice of predicate values, a variable \( x \) needs the value of variable \( y \), then the path of \( x \) is equal to \( \ldots + C \ o \ \{y, x\} + \ldots \).

2. the path of \( x \) is equal to \( \ldots + C \ o \ \{y\} + \ldots \) then, for the choice of predicate values corresponding to context \( C \), \( x \) needs the value of variable \( y \), or \( x = y \).

We first verify that the paths of variables \( x_1, \ldots, x_n \) in a statement \( x_1, \ldots, x_n = E \) not within a conditional branch follow the conditions given above. Based on the correctness of these paths, we conclude correctness of statements within conditional branches.

#### 4.3.1 Property of paths

For the moment, we consider statements which are not within branches of a conditional. Let us start by making an observation on the form of paths of outputs to expressions. Based on the different expressions in SubId, the expression which gives rise to different possible paths are conditionals, since there is one possible path corresponding to the consequent and one path corresponding to the alternative. Therefore, the paths of an expression containing different possibilities will have a path for when each predicate is false and for when each predicate is true. In canonical form, if the
path of a variable $x$ is $\text{context}_1 \circ \text{vars}_1 + \ldots + \text{context}_n \circ \text{vars}_n$, we know that any predicated set $\{y\}$ can be expanded to the form $\text{context}_1 \circ \{y\} + \ldots + \text{context}_n \circ \{y\}$ by using the rule of the predicated set algebra: $\{y\} = b_x \circ \{y\} + \overline{b}_x \circ \{y\}$ for any variable $y$ and predicate $x$. Therefore, each $\text{context}_i$ is not only mutually exclusive, but also exhaustive. With this property in mind, we can now step through the different possible expressions.

### 4.3.2 Simple expressions

**Constants**

The path of a constant is $\emptyset$ since a constant depends on nothing. Thus:

$$P[[CN_0]]Bf = (B, \emptyset, f)$$

**Correctness:**

For $x = CN_0$, $B(x) = \emptyset + \{x\} = \{x\}$. $x$ needs no variables, so condition (1) is trivially verified. The only variable in $x$’s path is $x$ so condition (2) is also verified.

**Variables**

To know the path of a variable $x'$, we need only look up the path associated with $x'$ in $B$, the path variable environment, therefore $P[[x']]Bf = (B, B(x'), f)$.

**Correctness:**

For $x = x'$, $B(x) = B(x') + \{x\}$. Since $x'$ must be within the scope of $x$, if $x$ is not in a conditional branch neither is $x'$. Therefore, we know from the preliminary observation that, if the canonical form of $x'$ is $\text{context}_1 \circ \text{vars}_1 + \ldots + \text{context}_n \circ \text{vars}_n$ then $x$ can be written in the following expanded form: $x = \text{context}_1 \circ \{x\} + \ldots + \text{context}_n \circ \{x\}$. Therefore, $B(x) = \text{context}_1 \circ (\text{vars}_1 + \{x\}) + \ldots + \text{context}_n \circ (\text{vars}_n + \{x\})$. For each $\text{context}_i$, $x$ needs the same variables as $x'$ included, and $B(x) = \ldots + \text{context}_i \circ \text{vars}_i + \ldots$. By induction, the variables needed by $x'$ within $\text{context}_i$ are $\text{vars}_i$, which includes $x'$ so condition (1) is verified. $B(x) = \ldots + \text{context}_i \circ (\text{vars}_i + \{x\}) + \ldots$ and $x$ needs
\( \text{vars}_i \) since \( x \) needs \( x' \) as well as the variables needed by \( x' \), so condition (2) is verified.

### 4.3.3 Primitive functions

The primitive functions in Subld are strict in all their arguments. Therefore the path of a primitive function is simply comprised of all of its arguments and all the variables that are needed to obtain the value of these arguments: \( P[[PF(x_1, \ldots, x_n)]]Bf = (B, B(x_1) + \ldots + B(x_n), f) \). From our preliminary observation we know that each \( B(x_i) \) can be rewritten in expanded form as \( B(x_i) = \text{context}_1 \circ \text{vars}_{i_1} + \ldots + \text{context}_m \circ \text{vars}_{i_m} \). Therefore, \( B(x_1) + \ldots + B(x_n) = \text{context}_1 \circ (\text{vars}_{i_1} + \ldots + \text{vars}_{n_1}) + \ldots + \text{context}_m \circ (\text{vars}_{i_m} + \ldots + \text{vars}_{n_m}) \).

Correctness, for \( x = PF(x_1, \ldots, x_n) \):

From our preliminary observation we know that each \( B(x_i) \) can be rewritten in expanded form as \( B(x_i) = \text{context}_1 \circ \text{vars}_{i_1} + \ldots + \text{context}_m \circ \text{vars}_{i_m} \). Therefore \( B(x_1) + \ldots + B(x_n) = \text{context}_1 \circ (\text{vars}_{i_1} + \ldots + \text{vars}_{n_1}) + \ldots + \text{context}_m \circ (\text{vars}_{i_m} + \ldots + \text{vars}_{n_m}) \).

Similarly, for each \( \text{context}_i \), \( x \) needs the variables \( x_1, \ldots, x_n \) as well as all the variables which they need, which are precisely \( \text{vars}_{i_1} + \ldots + \text{vars}_{i_m} \). Therefore, conditions (1) and (2) hold.

### 4.3.4 Conditionals

The abstraction rule for conditionals is as follows:

1. \( P[[\text{Cond}(x, E_t, E_f)]]Bf = \text{let} \ (B', p_1, \ldots, p_k, f) = P[[E_t]]Bf \)
2. \( r_1, \ldots, r_n = BV(E_t) \)
3. \( (B'', p'_1, \ldots, p'_k, f) = P[[E_f]]Bf \)
4. \( r'_1, \ldots, r'_m = BV(E_f) \)
5. \( B'' = B + \{r_1 = b_x \circ (B'(r_1) + x), \ldots, r_n = b_x \circ (B'(r_n) + x)\} \)
6. \( + \{r'_1 = \overline{b}_x \circ (B''(r'_1) + x), \ldots, r'_m = \overline{b}_x \circ (B''(r'_m) + x)\} \)
7. \( \text{in} \ (B'', b_x \circ (p_1 + \{x\}) + \overline{b}_x \circ (p'_1 + \{x\}), \ldots, b_x \circ (p_k + \{x\}) + \overline{b}_x \circ (p'_k + \{x\}), f) \)
To find the paths in a conditional we need to analyze the paths arising from the consequent and the alternative. This corresponds to lines (1) and (3). The path for the conditional expression is the tuple corresponding to the paths of each output of the conditional. Each output’s path is composed of the path resulting from the true branch when $x$ is true and the path resulting from the false branch when $x$ is false, as well as the variable $x$ in both of the possible paths (line 6).

Correctness, for $x_1, \ldots, x_k = Cond(x, E_t, E_f)$:

$$B(x_i) = b_x o (p_i + \{x\}) + \overline{b_x} o (p_i' + \{x\}).$$

For the context $b_x$, in effect $x_i = r_i$, and $B(x) = p_i + \{x\}$ where $B(r_i) = p_i$. We have already seen this case, therefore conditions (1) and (2) are verified. Note that since $p_i$ is of the form $context_1 o vars_1 + \ldots + context_n o vars_n$, $b_x o p_i = b o context_1 o vars_1 + \ldots + b_x o context_n o vars_n$.

Therefore, $b_x o context_i$ may not yield a plausible context, for example if $context_i = \overline{b_x}$. Our algebra allows for these impossible cases to be removed since $b_x o \overline{b_x} o P + P' = P'$. For example:

$$\{\text{if } x \text{ then } \{r = f(x, y) \text{ in } r\} \text{ else } z\}$$

where $f(a, b) = \{\text{if } a \text{ then } b \text{ else } 1\}$. The path of $r$ is:

$$b_x o (b_x o \{x, y, r\} + \overline{b_x} o \{x, r\})$$

$$= b_x o \{x, y, r\} + b_x o \overline{b_x} o \{x, r\}$$

$$= b_x o \{x, y, r\}$$

For the context $\overline{b_x}$, the same arguments hold.

### 4.3.5 User defined function calls

The rule for a function call is as follows:

1. $P[[F(a_1, \ldots, a_k)]]Bf = \lambda x_1 \ldots x_k.\text{paths} = f(F)$
2. $t_1, \ldots, t_m = IV(\lambda x_1 \ldots x_k.\text{path})$
3. $\text{paths}' = \text{paths}[t'_1/t_1, \ldots, t'_m/t_m, x_1/a_1, \ldots, x_k/a_k]$ (where $t'_i$ is fresh)
4. $S = \{a_1 = B(a_1), \ldots, a_k = B(a_k), t'_1 = \{\}, \ldots, t'_m = \{\}\}$
5. in $(B, S \text{ paths}', f)$
When a function call is encountered, the assumption is that the function has already been analyzed and therefore its path scheme is in \( f \). We therefore use this path scheme to deduce the path of call (line 1). The path of a function call is the tuple corresponding to the paths of the outputs of the call. We can therefore use the information contained in the path scheme about the paths of the outputs of the function with a few modifications. We must substitute the formal parameters with the actual parameters, as well as all the variables leading to the inputs to know the variables that lead to the outputs in lines (3), (4). To insure that the internal variables appearing in the output paths do not clash across calls to the same function, they are renamed to fresh variables each time (line 3).

However, why even keep information about internal variables since these have no meaning outside of the call? Internal variables are important as a way of labeling the different contexts appearing in the output paths. Thus, if internal variables are consistently renamed across all the output paths of a same call, we have preserved all the information needed as we will see in the next chapter. We are able to tell which paths lead to each of the outputs within the same context. The other important information that is preserved through consistent renaming is the relation of one context to another. If a context \( b_\nu \) and a context \( \overline{b_\nu} \) are consistently renamed to say \( b_{\nu'} \) and \( \overline{b_{\nu'}} \), after renaming we still know that one context is the opposite case of the other. The latter information plays a role in our algebra, most importantly when taking the intersection or union of two predicated sets, by enabling elimination of cross terms containing conflicting context information.

Therefore, although predicate renaming causes our predicate domain to become infinite for our analysis, we cannot resort to somehow collapsing the contexts contained in paths. However, renamed internal variables bring nothing to our analysis outside of their role in contexts, therefore they are substituted with the empty path (line 4) to remove their appearance everywhere except within the contexts.

For example, if \( f \) has as path scheme \( \lambda x \ y. b_t o \{ x, t \} + \overline{b_t} o \{ t, y \} \) and \( f \) is called on \( a \) with path \( \{ z, w, a \} \) and \( c \) with path \( \{ c \} \), the path of the \( f(a, b) \) is \( b_{\nu'} o \{ a, z, w \} + \overline{b_{\nu'}} o \{ c \} \) where \( t' \) is a fresh variable.
Correctness, for $x_1, \ldots, x_n = F(a_1, \ldots, a_k)$:

The correctness stems from the substitution $S$. If for a given context $C$, the output $o$ of $F$ needs the input variables $a_i, \ldots, a_j$, then from previous cases we know that $o$'s path is $B(a_i) + \ldots + B(a_j)$, which is exactly what is produced by $S \{a_i, \ldots, a_j\}$. Therefore, conditions (1) and (2) are verified.

4.3.6 Statements within branches of a conditional

We recall the rule for conditionals:

(1) $P[[\text{Cond}(x, E_t, E_f)]]Bf = \text{let } (B', p_1, \ldots, p_k, f) = P[[E_t]]Bf$
(2) $r_1, \ldots, r_n = BV(E_t)$
(3) $(B'', p'_1, \ldots, p'_k, f) = P[[E_f]]Bf$
(4) $r'_1, \ldots, r'_m = BV(E_f)$
(5) $B''' = B + \{r_1 = b_x o (B'(r_1) + x), \ldots, r_n = b_x o (B'(r_n) + x)\}$
(6) $\quad + \{r'_1 = \overline{b_x} o (B''(r'_1) + x), \ldots, r'_m = \overline{b_x} o (B''(r'_m) + x)\}$
(7) $\quad \text{in } (B''', b_x o (p_1 + \{x\}) + \overline{b_x} o (p'_1 + \{x\}),$
$\quad \quad \ldots, b_x o (p_k + \{x\}) + \overline{b_x} o (p'_k + \{x\}), f)$

For statements within the branch of a conditional, the paths of the variables are first found with respect to the block corresponding to the conditional branch, $E_t$ or $E_f$ in lines (2) and (4). We have already seen that with respect to blocks, the paths of variables verify conditions (1) and (2). However, since this block occurs within a conditional, each variable also needs the value of the predicate $x$ so $x$ is added to the path of each variable for each context (from the preliminary observation) (line 5). Since these paths only make sense within the true or false branch of a conditional, we add $b_x$ or $\overline{b_x}$ to each context (line 5). We assume that dead code has been eliminated prior to our analysis, and that therefore we will not have the situation:

```latex
{ \text{if } y \text{ then } \{\ldots\} }
\quad \text{else } \{ \text{if } y \!
\quad \quad \text{then } \{ x = \ldots \} }
```
else \{\ldots\}\}

which, might yield $B(x) = b_x \circ \overline{b_x} \circ \text{vars}$, a path which makes no sense, since it cannot occur. If such a possibility occurs within a sum of various other possibilities, we have already seen these can be eliminated through the algebra.

### 4.3.7 Function definitions

Here is the rule for analyzing a block at the top level:

$$A[[F(x_1, \ldots, x_n) = \{S \in y_l, \ldots, y_k\}]f = \text{let } (B', p_1, \ldots, p_k, f) =$$

$$P[[\{S \in y_l, \ldots, y_k\}][x_1 = x_1, \ldots, x_n = x_n]f$$

$$f' = f + \{F : \lambda x_1 \ldots x_n. p_1, \ldots, p_k\}$$

in $A[[F(x_1, \ldots, x_n) = \{S \in y_l, \ldots, y_k\}]f'$

Before analyzing $F$, the function path environment associates $\bot$ with the path of each output of $F$. To find the paths of all the variables in the block and find the paths of the outputs to the block, we call $P$ on the body of the block. $f$ is then augmented by adding the new path we just obtained for $F$, and recursively analyzing the body until we reach the termination condition (a safe approximation for $F$).

#### Termination

The termination condition is not to reach a fixed point over the function path scheme of $F$ since recursive function calls introduce new variables at each iteration as a result of the function call rule: internal variables are renamed each time we analyze the function call. This means that the contexts contained in a path will keep on growing infinitely. Therefore we introduced a special termination condition.

Proof of termination (from [5]).

Since the number of variables (though not the number of predicate labels) is bounded by the number of identifiers in the function being analyzed, there is a strict upper bound on the result of $t(p_i)$. Thus, to prove termination we establish that $t(p_i^h) \subseteq t(p_i^{h+1})$ for every $h$ and its corresponding $p_i$. 

42
The program syntax is fixed, and thus at each iteration the abstraction rules will give rise to the same equations between paths. A path can only change if the path of a variable used to construct it has also been modified. These changes only occur (transitively) as a result of the instantiation of a recursive function, $h$, whose output paths have changed. So if we assume by induction that $t(p_i^{k-1}) \subseteq t(p_i^k)$, then by induction (and monotonicity of our algebra) when we instantiate $h$ on the next iteration $t(p_i^k) \subseteq t(p_i^{k+1})$. Note that no collapsing can occur which might cause $t(p_i^{k+1}) \subset t(p_i^k)$ since an instantiation step introduces fresh predicate names to a previously collapsed term.

**Correctness**

The correctness conditions (1) and (2) only make sense within each individual iteration. Given that we examined all possible expressions, we are already assured that these hold.

The correctness that needs to be addressed is the safety of the termination condition for the fixed point iteration. Safety, in this case, relies on how paths will be used in our analysis. We therefore defer this proof to the section arguing correctness of the algorithm as a whole.

**An example**

\[
\text{sw a b c} = \{ r = \{ \text{if a} } \\
\text{ then } \{ t_1 = false; } \\
\text{ t_2 = sw a b c; } \\
\text{ t_3 = 1 + t_2; } \\
\text{ in } \\
\text{ t_3} \}
\]

\[
\text{else 0 } \}
\]

\[
\text{in r } \}
\]

Fixed pointing over the paths for the function switch yields the following paths:
<table>
<thead>
<tr>
<th>Iter</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$b_x \circ {x, t_1}$</td>
<td>$b_x \circ {x, t_1}$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$b_x \circ \bot$</td>
<td>$b_x \circ b_y \circ \bot + b_x \circ b_y \circ {y, x, t_1, t_2}$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$b_x \circ \bot$</td>
<td>$b_x \circ b_y \circ \bot + b_x \circ b_y \circ {y, x, t_1, t_2, t_3}$</td>
</tr>
<tr>
<td>r</td>
<td>$b_x \circ \bot + b_x \circ {x, r}$</td>
<td>$b_x \circ b_y \circ \bot + b_x \circ b_y \circ {y, x, t_1, t_2, t_3, r}$</td>
</tr>
<tr>
<td>sw</td>
<td>$b_x \circ \bot + b_x \circ {x, r}$</td>
<td>$b_x \circ b_y \circ \bot + b_x \circ b_y \circ {y, x, t_1, t_2, t_3, r}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$b_x \circ {x, t_1}$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$b_x \circ b_y \circ b_x \circ \bot + b_x \circ b_y \circ b_x \circ {x, y, z, t_1, t_2} + b_x \circ b_y \circ {x, y, t_1, t_2}$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$b_x \circ b_y \circ b_x \circ \bot + b_x \circ b_y \circ b_x \circ {x, y, z, t_1, t_2, t_3} + b_x \circ b_y \circ {x, y, t_1, t_2, t_3}$</td>
</tr>
<tr>
<td>r</td>
<td>$b_x \circ b_y \circ b_x \circ \bot + b_x \circ b_y \circ b_x \circ {x, y, z, t_1, t_2, t_3}$ $+ b_x \circ b_y \circ {x, t_1, t_2, t_3, r} + b_x \circ {x, r}$</td>
</tr>
<tr>
<td>sw</td>
<td>$b_x \circ b_y \circ b_x \circ \bot + b_x \circ b_y \circ b_x \circ {x, y, z, t_1, t_2, t_3, r}$ $+ b_x \circ b_y \circ {x, t_1, t_2, t_3, r} + b_x \circ {x, r}$</td>
</tr>
</tbody>
</table>

The last two iterations, verify the termination condition as the different possible sets of variables have not changed: $\bot, \{c, b, a\}, \{b, a\}, \{a\}$. 
<table>
<thead>
<tr>
<th>Iter</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$b_x \circ {x, t_1}$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ {t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ \overline{b_z} \circ {t_2, t_1, z, y, x} + b_x \circ b_y \circ {t_1, t_2, y, x}$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ {t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ \overline{b_z} \circ {t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ {t_3, t_1, t_2, y, x}$</td>
</tr>
<tr>
<td>r</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ {r, t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ {r, t_3, t_1, t_2, y, x}$ $+ b_x \circ \overline{b_z} \circ {x, r}$</td>
</tr>
<tr>
<td>sw</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ {r, t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ {r, t_3, t_1, t_2, y, x}$ $+ b_x \circ \overline{b_z} \circ {x, r}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$b_x \circ {x, t_1}$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {t_2, t_1, z, y, x}$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {t_3, t_2, t_1, z, y, x}$</td>
</tr>
<tr>
<td>r</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {r, t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x}$</td>
</tr>
<tr>
<td>sw</td>
<td>$b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ \bot + b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {r, t_3, t_2, t_1, z, y, x}$ $+ b_x \circ b_y \circ b_z \circ b_{t_1} \circ b_{t_2} \circ b_{t_3} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x} + b_x \circ b_y \circ \overline{b_z} \circ {r, t_3, t_2, t_1, z, y, x}$</td>
</tr>
</tbody>
</table>
Chapter 5

Dependence, Demand and Tolerance Labels

From the abstract interpretation step, we have obtained dependence information for each variable directly from the syntax. With this information we can compute dependence, demand, and tolerance labels for each variable. These labels will be used to determine which variables may be put in the same partition.

5.1 Definitions

The following definitions and comments come from [5].

Definition 10 (Dependence) The dependence label of a variable \( x \) is the predicated set of all the input variables \( x \) depends on. Dependence labels can be computed very simply given the path variable environment \( B \) and the set of locally bound variables (non-parameters) \( V \):

\[
Dep(x) = B(x)[\{\}/V, \{\}/\perp]
\]

Note that we eliminate bottoms as we compute partitions; in previous analyses (which were done case-by-case), bottom nodes were simply ignored during partitioning. Thus only partitions obtained for non-bottom dependencies matter.
Definition 11 (Demand) In a block, the demand label of a variable $x$ is the predicated path of all outputs which depend on $x$. Demand labels can be computed given the path variable environment $B$ and a list of result variable names $R$ as follows:

$$\text{Dem}(x) = \bigcup_{r \in R} (B(r) \cap \{x\})[r/x, \{\}/\perp]$$

Computing demand essentially reverses dependencies; $B(r) \cap \{x\}$ selects outputs which depend on $x$ and the substitution indicates that $x$ will be demanded by them under the same circumstances.

Definition 12 (Tolerance) An output $r$ tolerates a variable $x$ when a dependency on $x$ can be added to $r$ in $B$ without affecting the input/output dependencies of $C$. The tolerance label of a variable $x$ is the predicated path of outputs which tolerate $x$. Tolerance labels can be computed as follows:

1. $\text{Tol}(i) = \text{Dem}(i), i \in I$
2. $\text{Tol}(x) = \text{Tol}'(\text{Dep}(x))$

$$\text{Tol}'(\text{vars}) = \bigcap_{i \in \text{vars}} \text{Dem}(i)$$
$$\text{Tol}'(b \circ p) = b \circ \text{Tol}'(p)$$
$$\text{Tol}'(p_1 + p_2) = \text{Tol}'(p_1) \cap_p \text{Tol}'(p_2)$$

5.2 Correctness of Labels for Basic Blocks

In [2], dependence, demand and tolerance sets are computed for a variable within the context of a basic block. These sets are computed using a dependence graph corresponding to the basic block in question. The dependence graph of a basic block is defined as follows:

Definition 13 (Dependence Graph) A dependence graph of a basic block $BB$ is the directed graph $G = (V, E)$ where $V$ is the set of variables occurring in $BB$, and $(x, y) \in E$ if there is a statement such that $x = \text{PF}(\ldots, y, \ldots)$. 
The definition of dependence, demand and tolerance sets are as follows:

**Definition 14 (Dependence)** The dependence set of a node in the basic block $BB$ is the set of inlets on which it depends.

$Dep(i) = \{i\}$ if $i$ is an input node.

$Dep(n) = \cup_{(m,n) \in BB} Dep(m)$

From this definition, we can make the following observation:

**Observation 1** An input $i$ is in a node $n$’s dependence set if and only if there is a path from $i$ to $n$ in $BB$.

**Induction Proof:**

**Base Case:** An input $i$ is the only node in $i$’s dependence set.

**Induction Step:** A node $n$ of depth $k$ has input $i$ in its dependence set if and only if there is a path from $i$ to $n$ in $BB$. Let us show the same property holds for a node $n'$ of depth $k + 1$.

$\Leftarrow$ If there is a path from $i$ to $n'$, then there exists a path from $i$ to one of the parents of $n'$. Call this parent $n$. The node $i$ is in the dependence set of $n$ (induction hypothesis) and therefore in the dependence set of $n'$ (recursive definition).

$\Rightarrow$ Any node $n'$ that has $i$ in its dependence set must be a child of a node $n$ having $i$ in its dependence set (follows from recursive definition). Since $n$ has $i$ in its dependence set, there is a path from $i$ to $n$ (induction hypothesis), and therefore a path from $i$ to $n'$ since $n'$ is the child of $n$.

To compute the dependence predicated set of a variable $x$, non-input variables are removed from the path of $x$. Therefore, for basic blocks, since there is only one possible path, the dependence predicated sets we defined earlier contain the same information as dependence sets.

A corollary to the previous observation is:

**Observation 2** An input $i$ is in an output $o$’s dependence set if and only if there exists a path from $i$ to $o$. 

48
Definition 15 (Demand)  The demand set of a node in a basic block $BB$ is the set of outlets which depend on it.

$Dem(n) = \{o\}$ if $o$ is an output node

$Dem(n) = \cup_{(n, m) \in BB} Dem(m)$

Observation 3  An output $o$ is in a node $n$'s demand set if and only if there is a path from $n$ to $o$ in $BB$.

Induction Proof:

Base Case: An output $o$ is the only node in $o$'s demand set.

Induction Step: A node $n$ of depth $k$ has output $o$ in its demand set if and only if there is a path from $n$ to $o$ in $BB$. Let us show the same property holds for a node $n'$ of depth $k - 1$.

$\Leftarrow$ If there is a path from $n'$ to $o$, then there exists a path from one of the children of $n'$ to $o$. Call this child $n$. The node $o$ is in the demand set of $n$ (induction hypothesis) and therefore in the demand set of $n'$ (recursive definition).

$\Rightarrow$ A node $n'$ that has $o$ in its demand set must be a parent of a node $n$ having $o$ in its demand set (follows from the recursive definition). Since $n$ has $o$ in its demand set, there is a path from $n$ to $o$ (induction hypothesis), and therefore a path from $n'$ to $o$ since $n'$ is the parent of $n$.

A corollary to the previous observation:

Observation 4  An output $o$ is in an input $i$'s demand set if and only if there exists a path from $i$ to $o$.

From our observations on dependence and demand sets, we make the following observations:

Observation 5  Input $i$ is in an output $o$'s dependence set if and only if output $o$ is in input $i$'s demand set.

Observation 6  If node $n$ is on a path from input node $i$ to output node $o$ then $i$ is in $n$'s dependence set, and $o$ is in $n$'s demand set.
From the previous observation, we can also see that demand predicate sets in the case of basic blocks contain the same information as demand sets.

**Definition 16 (Tolerance)** An output $o$ tolerates a node $n$ if and only if an edge $(n, o)$ can be added to $BB$ without affecting input-output connectivity. The tolerance set of a node $n$ is the set of outputs that tolerate $n$.

$Tol(o) = Dem(o)$ if $o$ is an output.

$Tol(n) = \cap_{(m, n) \in BB} Dem(n)$

To see how our definition of tolerance sets is related to tolerance labels, let us give another recursive definition.

**Observation 7** Another way to compute tolerance sets is:

$Tol(n) = Dem(n)$ if $n$ is an input node.

$Tol(n) = \cap_{i \in Dep(n)} Dem(i)$

Induction Proof:

Base Case: For an input node $i$, $Tol(i) = Dem(i)$.

Induction Step: A node $n$ of depth $k$ has a tolerance set $Tol(n) = \cap_{i \in Dep(n)} Dem(i)$. Let us show that the same equation holds for a node $n'$ of depth $k + 1$.

$$Tol(n') = \cap_{(m, n) \in BB} Tol(m')$$

$$= \cap_{(m', n') \in BB}(\cap_{i \in Dep(m')} Dem(i))$$

$$= \cap_{i \in Dep(m') \land (m', n') \in BB} Dem(i)$$

$$= \cap_{i \in Dep(n)} Dem(i)$$

since $i$ such that $i \in Dep(m')$ and $(m', n') \in BB$

$\Rightarrow i$ such that there is a path from $i$ to $m'$ in $BB$ and $(m', n') \in BB$

$\Rightarrow i$ such that there is a path from $i$ to $n'$

$\Rightarrow i$ such that $i \in Dep(n)$
This observation shows us that the tolerance predicated sets also contain equivalent information to the tolerance sets.

5.3 Observations for General Blocks

These observations will become useful when we prove correctness of our analysis. A general block contains conditionals as well as function calls. Therefore, according to what boolean values we associate with each predicate in the general block, we will obtain different basic block corresponding to each possible control flow. Given a label for a general block, the observations show that the correct label can be obtained for every derived basic block.

In the following observations, a context corresponds to: $C ::= \text{pred|pred} \circ o C$.

Observation 8 Let Block be the basic block obtained by assigning a boolean value to each of the predicate nodes appearing in the general block $G$. Let $C$ be the context corresponding to this choice of predicate values. Let $x$ be a variable in $G$ and $\text{Dep}(x)$ be its dependence label.

A variable $x$ in Block has input variable $i$ in its dependence label if and only if in the general block $\text{Dep}(x) = \ldots + C \circ \{i\} + \ldots$

Proof. If variable $x$ is an input variable, trivial. From the correctness of paths and labels for basic blocks we argue the following:

$\Rightarrow$ If variable $x$ is not an input variable and has $i$ in its dependence label in Block, then the path of $x$ is $\{\ldots, x', \ldots\}$ where the path of $x'$ in Block is $\{\ldots, i, \ldots\}$. This means that for $G$ within the scope of the context $C$, variable $x$ needs $x'$, as well as the variables needed by $x'$ in the same context, and therefore the path of $x$ is $\ldots + C \circ \{x', i\} + \ldots$

$\Leftarrow$ If variable $x$ is not an input variable and the dependence label of $x$ in $G$ is $\ldots + C \circ \{i\} + \ldots$, then within the context $C$, $x$ needs a variable $x'$ which within context $C$ needs $i$. Therefore, in Block, the path of $x$ is $\{\ldots, x', i, \ldots\}$ which means the dependence label of $x$ is $\{\ldots, i, \ldots\}$.
Observation 9  Let Blck be the basic block obtained by assigning a boolean value to each of the predicate nodes appearing in the path of the variables in the general block G. Let C be the context corresponding to this choice of predicate values. Let x be a variable in G and Dem(n) be its demand label.

A variable x in Blck has an output variable o in its demand label if and only if in the general block Dem(x) = \ldots + C o \{o\} + \ldots.

Proof.

⇒ If variable x in Blck has an output variable o in its demand label, output o has variable x in its path. This means that for G within the scope of the context C, o's path is \ldots + C o \{x\} + \ldots. Therefore from the definition of demand labels, Dem(x) = \ldots + C o \{o\} + \ldots.

⇐ If Dem(x) = \ldots + C o \{o\} + \ldots, then the path of o is \ldots + C o \{x\} + \ldots. Therefore, in Blck, the path of o is \{\ldots, x, \ldots\}. From this we deduce that the demand label of x in Blck is \{\ldots, o, \ldots\}.

The correctness of tolerance labels follows from the correctness of dependence and demand labels.
Chapter 6

From Source Code to Partitioned Code

In this chapter, all the topics presented in the previous chapters come together: Predicated sets are used to compute paths for every variable in a block. From the path information dependence, demand, and tolerance labels are computed for each variable. Finally, variables are put in the same partition according to these labels and partitioned code is generated. In this chapter we describe the algorithm and argue its correctness.

6.1 Partitioning

A partitioning of a block consists of grouping statements into threads with the following properties:

1. Instructions within a thread can be ordered at compile time such that the ordering is correct for all contexts in which the procedure may be invoked.

2. It must be possible for values computed in an executing thread to not be visible to other threads before the thread completes.

Since the source code is topologically sorted, there are no cyclic dependencies. Furthermore, values computed in a thread once spawned cannot be used until the
thread completes. Therefore two threads cannot depend on each others’ results, and we are sure that instructions within a thread can be topologically sorted.

### 6.2 Partitioned Code Grammar

#### \( Toplevel' \) :

\[
F(x_1, \ldots, x_k) = \{x_1, \ldots, x_k = spawn E in B'\}
\]

\[
| F(x_1, \ldots, x_k) = \{x_1, \ldots, x_k = E in B'\}
\]

\[
B' = \{x_1, \ldots, x_k = spawn E in B\}
\]

\[
| \{x_1, \ldots, x_k = E in B\}
\]

\[
| r_1, \ldots, r_k
\]

\[
E' := SE
\]

\[
| PF(SE_1, \ldots, SE_k)
\]

\[
| Cond(SE, E_t, E_f)
\]

\[
| B'
\]

\[
| F(x_1, \ldots, x_k)
\]

\[
SE := x | V
\]

\[
V := CN_o
\]

### 6.3 Algorithm

The rewrite rules from chapter one have been already applied to the topologically sorted source code we are starting from.

#### 6.3.1 Compute labels necessary for partitioning

1. Assign a number to each statement according to the topologically sorted order of the statements in the block (do this recursively for each sub-block as well).

2. In a forward pass, compute the paths for each variable.
3. From the path information, compute the dependence, demand, and tolerance sets for each variable

4. Do until no labels change:

   (a) Merge variables with the same demand label: For variables $x_1, \ldots, x_n$ having the same demand label, compute a new dependence label $Dep'$ as follows:

   $$Dep'(x_1) = \ldots = Dep'(x_n) = Dep(x_1) \cup \ldots \cup Dep(x_n)$$

   and for all other variables compute their new dependence label as follows:

   $$Dep'(y) = (Dep(y) \cap \{x_1, \ldots, x_n\})[x_i/Dep'(x_i)] \cup Dep(y)$$

   Using the new dependency labels for each variable, compute the new tolerance label.

   (b) Merge variables with the same tolerance label: For variable $x_1, \ldots, x_n$ having the same tolerance label, compute a new tolerance label $Tol'$ as follows:

   first compute the new demand label:

   $$Dem'(x_1) = \ldots = Dem'(x_n) = Dem(x_1) \cup \ldots \cup Dem(x_n)$$

   and their new $Dep'$ label:

   $$Dep'(x_1) = \ldots = Dep'(x_n) = Dep(x_1) \cup \ldots \cup Dep(x_n)$$

   and from this compute their new tolerance label.

   For all other nodes compute their new dependence label as follows:

   $$Dep'(y) = (Dep(y) \cap \{x_1, \ldots, x_n\})[x_i/Dep'(x_i)] \cup Dep(y)$$

   and their new demand labels as follows:

   $$Dem'(y) = (Dem(y) \cap \{x_1, \ldots, x_n\})[x_i/Dem'(x_i)] \cup Dem(y)$$

   From this derive their new tolerance label.

6.3.2 Generate partitioned code from labels

1. Using these labels, and the number assigned to each statement according to the topological sort, produce partitioned code:
For each block and sub-block, go through the statements in the block in order, grouping them together as follows:

Let $S_i$ be $x_1, \ldots, x_n = E$ and $S_j$ be $y_1, \ldots, y_k = E'$, and $i < j$. If $S_i$ and $S_j$ have the same dependence label, and there is no statement $S_k$ in the source code such that $i < k < j$ where $S_k$ is a statement with multiple variables on its left hand side who do not have the same dependence labels, group $S_i$ and $S_j$ into one statement as follows:

\[
x_1, \ldots, x_n, y_1, \ldots, y_k = \{ r_1, \ldots, r_n = E \\
in \{ r'_1, \ldots, r'_k = E' \\
in r_1, \ldots, r_n, r'_1, \ldots, r'_k \} \}
\]

and continue down the rest of the statements with the modified block.

2. Once code has been rearranged as above, convert the modified source code to partitioned code using the rules given below.

**Rules for converting from source code to partitioned code:**

\[
p : v_1 \times \ldots \times v_n \rightarrow \text{Bool}
\]

$v_k$: is a variable.

\[
B : B \rightarrow p \rightarrow B'
\]

\[
E : E \rightarrow p \rightarrow E'
\]

\[
S : S \rightarrow p \rightarrow S'
\]

\[
T : \text{TopL} \rightarrow p \rightarrow \text{TopL}'
\]

where $S = x_1, \ldots, x_n = E$

and $S' = x_1, \ldots, x_n = \text{spawn } E' \mid x_1, \ldots, x_n = E'$

$p$, when given a set of variables, returns true if they are in the same partition, otherwise returns false.

\[
S[[x = SE]]p := x = SE
\]

\[
S[[\{x_1, \ldots, x_k = PF(SE_1, \ldots, SE_k)\in B\}]]p := \{x_1, \ldots, x_k = PF(SE_1, \ldots, SE_k)\in B\}
\]
\[ S[[x_1, \ldots, x_k = \text{Cond}(SE, E_t, E_f)]]p := \text{if } p(x_1, \ldots, p_k) \]
\[ \text{then } x_1, \ldots, x_k = \text{spawn Cond}(SE, E[[E_t]]p, E[[E_t]]p) \]
\[ \text{else } x_1, \ldots, x_k = \text{Cond}(SE, E[[E_t]]p, E[[E_t]]p) \]
\[ S[[x_1, \ldots, x_k = F(y_1, \ldots, y_n)]]p := \text{if } p(x_1, \ldots, p_k) \]
\[ x_1, \ldots, x_k = \text{spawn } F(y_1, \ldots, y_n) \]
\[ x_1, \ldots, x_k = F(y_1, \ldots, y_n) \]
\[ S[[x_1, \ldots, x_k = B']]p := x_1, \ldots, x_k = \text{spawn } B[[B']] \]
\[ E[[SE]]p = SE \]
\[ E[[PF(y_1, \ldots, y_n)]]p := PF(y_1, \ldots, y_n) \]
\[ E[[F(y_1, \ldots, y_n)]]p := F(y_1, \ldots, y_n) \]
\[ E[[B']]p := B[[B']]p \]
\[ B[[\{x_1, \ldots, x_k \text{ in } B\}]]p = \{S[[x_1, \ldots, x_k = E]]p \text{ in } B[[B]]p\} \]
\[ B[[\{x_1, \ldots, x_k \text{ in } r_1, \ldots, r_k\}]]p = \{S[[x_1, \ldots, x_k = E]]p \text{ in } r_1, \ldots, r_k\} \]
\[ TopL[[F(x_1, \ldots, x_k) = \{x_1, \ldots, x_k = E \text{ in } B\}]]p = B[[\{x_1, \ldots, x_k = E \text{ in } B\}]]p \]

### 6.4 Correctness

Our algorithm is exactly the same as Coorg’s algorithm except that instead of using dependence, demand, and tolerance sets, we use predicated sets to compute dependence, demand, and tolerance labels. We therefore need to prove that using these labels is safe. The labels are derived from paths which we have already proved correct, except for the fixed point termination condition. We therefore start by proving safety of the termination condition, and then argue that the partitions obtained using paths are correct.

#### 6.4.1 Safety of termination condition

The correctness for the iteration depends on our use of the resulting predicated paths. Our analysis does not need a fixed point to be reached on the predicated sets we obtain. Rather, we need to show that demand, dependence and tolerance sets (labels)
will yield the same partitions even if we iterate beyond termination. This is equivalent
to establishing that two variables $x_1$ and $x_2$ will have identical dependence, demand,
or tolerance labels once termination is reached if and only if they have identical
dependence, demand, or tolerance labels in subsequent iterations.

Our proof of termination showed that a fixed point must be reached on the possible
disjoint dependencies (the values of $t$). What will continue to change with further
iteration after termination are the contexts within which these dependencies occur.
Since the syntax has not changed, and the termination condition is verified, the same
sets of variables will continue to appear in the path information of a variable for all
future iterations. Therefore, if the context of variable $x$ changes then it will affect all
identical labels in an identical way if the corresponding paths contain $x$.

Because we are only adding new predicate labels (and already did so for at least
one iteration), no identical labels will become different as a result of the change in
$x$. Thus, although paths will continue to change between iterations, if two variables
have the same label at termination they will continue to do so, and if their labels
differ they will remain different.

\subsection{Correction of partitions}

To show the correctness of the partitions we obtain, we rely on the basic block cor-
rectness criterion defined by Coorg:
Let $Blck$ be a basic block. The partitioned block is correct if and only if acyclicity
and input/output connectivity are preserved. This means:

1. No threads in the partitioned block can be mutually dependent on each other.

2. Let $i$ be an input of $Blck$ and $o$ an output. $Dem(i)$ (respectively $Dep(o)$) must
   be invariant throughout the fixed pointing on labels.

The latter condition, stems from the fact that if $Dep(o)$ were to change, then an
input must have been added or removed from the path of $o$, which would mean that
input/output connectivity changed. The same argument holds true for $Dem(i)$. 
Given a basic block, we know that this criterion holds, since we have shown that in the case of basic blocks the demand, dependence, and tolerance labels are identical to those used in Coorg’s thesis.

From the correctness criterion for basic blocks stems the criterion for general blocks also defined by Coorg: partitions must be safe with respect to all the basic blocks obtained from every possible dynamic unfoldings of computation.

**Proof**

Indeed, if a fixed set of boolean values is associated with the predicate variables, and the block is unfolded, the demand and dependence labels we would obtain for the context corresponding to the values attributed to the predicates are the same. This could be done for all possible combinations of values that can be attributed to predicates. As we only put two variables in the same partition when it is possible to do so in every case, we are guaranteed to have safe partitions for every possible execution. Note that unfolding a recursion too far gives us no additional information—safety shows that the partitions of the outer scope will not change regardless of how much we unfold the inner recursion.

However, when producing partitioned code, statements, not variables are grouped together. Therefore, although two variables \( x \) and \( y \) can be in the same partition, two statements \( \ldots, x, \ldots = E \) and \( \ldots, y, \ldots = E' \) may not necessarily be in the same thread. This is a result of the expressions having multiple outputs. Two statements can only be in the same thread, if all the variables on the LHS have the same label. Let us consider an example:

\[
\{x_1 = a + 5 \in \{y_1, y_2 = f(x_1, b) \in \{x_3 = y_1 + 2, \ldots\}\}\}
\]

and \( f(x, y) = \{x' = 2 \times x \in x', y\} \).
Clearly, $x_1, y_1, x_3$ could go in the same partition. However, $y_1, y_2$ are outputs of the function call $f$, therefore we cannot put $y_1$ in one partition with other variables and $y_2$ in a different partition with other variables. Therefore, we are left with $x_1$ and $x_3$ spawned off as a thread $t_1$ in one partition, and the computation of $f$ spawned off as a separate thread, $t_2$. However, what if $x_3$ gets fed back into input $b$? Then we would have to interrupt $t_1$ after getting the value for $x_1$, then start $t_2$ to get the value of $y_1$, then interrupt $t_2$ and go back to $t_1$ to get the value of $x_3$, and then go back to $t_2$ to get the value of $y_2$. We do not want such situations, where two threads have to be interrupted to compute each others values. This is why we sometimes have to break partitions into smaller ones, so $x_1$ and $x_3$ must be in separate threads. Thus acyclicity is preserved.
Chapter 7

Conclusion

In this thesis, we have presented a partitioning algorithm for a first order, functional language with simple data types. Our algorithm is completely syntax directed for basic blocks, as well as general blocks. It does not depend on an intermediate dataflow representation and does not use a dependence graph. This means we have a way to systematically treat conditionals, as well as function calls.

Our algorithm uses predicated sets to capture dependence information which we use to do partitioning. Unlike previous algorithms, we only need to make one forward pass through the source code to obtain all the dependence labels. To compute dependence, demand, and tolerance sets, a forward and a backward pass is necessary. Also, predicated sets allow us to group information about every possible control flow in one representation, such that each step of our algorithm takes into consideration every possibility at least once. Previously, the algorithm entailed conducting an analysis for each possibility independently.

In this chapter we conclude by outlining further possible areas of research.

7.1 Implementation

We have only presented the algorithm in this thesis. The next step will be to implement the algorithm and compare it with previous ones. This will involve finding a way to efficiently perform operations on predicated sets, notably the intersection
operation. The trivial way of taking the intersection of two predicated sets is by expanding the sets, such that both predicated sets have the maximum number of identical contexts. Then for the contexts that are the same, one can simply take set intersection of the sets of variables which are associated with those contexts. For example:

\[ \{x\} \cap_p (b_t \circ \{x, y\} + \overline{b_t} \circ \{y, z\}) \]
\[ = (b_t \circ \{x\} + \overline{b_t} \circ \{x\}) \cap_p (b_t \circ \{x, y\} + \overline{b_t} \circ \{y, z\}) \]
\[ = (b_t \circ \{x\} \cap_p \{x, y\}) + \overline{b_t} \circ \{x \cap_p \{y, z\}\} \]
\[ = b_t \circ \{x\} + \overline{b_t} \circ \{\} \]

If a block contains \(n\) different predicates in the path of its outputs, then this means there are \(2^n\) different possible contexts which makes expanding predicated sets costly. If predicated sets are not expanded because of the rules of our algebra, they can be collapsed and represented in a compact form.

### 7.2 Recursive Analysis

A shortcoming of this thesis is the unconventional termination rule needed for the fixed point iteration to obtain paths. The problem with the way we use predicated sets is that the domain becomes infinite as we rename the internal variables for each separate call to a function. It would therefore be interesting to find a way to collapse the paths in a way that would enable the fixed pointing to end naturally.

### 7.3 Extensions

#### 7.3.1 Possible indirect extensions

The complexity of the let block analysis allows us to indirectly extend our analysis to some interesting cases, such as letrec blocks, as well as blocks containing calls to an unknown function.
Letrec blocks

The analysis can be extended to a letrec block by indirectly converting it to a let block, that is by lifting out its internal cycles prior to performing the partitioning. The algorithm to lift cycles out of a letrec block is described in [2]. The algorithm uses the dependence graph corresponding to the letrec block to compute its minimal feedback set. The variables in the feedback set $V$ are substituted with fresh variables $V'$ in every right hand side of all statements in the block. The variables in $V$ are then added to the set of outputs and the variables in $V'$ to the set of inputs. For example,

$$h() = \{ x, z = f(y) \}
\quad y = g(z)
\quad \text{in } x\}$$

has as possible minimal feedback set $\{y\}$. Let $w$ be a fresh variable. $w$ is substituted for $y$ on the RHS of every statement. Then $y$ is added to the outputs, and $w$ to the inputs.

$$h(w) = \{ x, z = f(w); \}
\quad y = g(x); 
\quad \text{in } x, y\}$$

By applying the partitioning analysis indirectly to a transformed letrec block, correct but non-optimal partitions are generated. The partitions are correct for any possible feedback from an output to an input, rather than only for the specific output to input feedback arising from the internal cycle in the letrec block. Continuing with the above example, the partition will be correct if $x$ or $y$ feeds back into $w$. In the original letrec block there was only one cycle, corresponding to a feedback from $y$ to $w$. Although our analysis does not handle letrec blocks directly, it gains in simplicity since we avoid the tricky situation involving computing cyclic dependencies.

Function calls

If no information about a function $f$ is known, we can still produce a correct partitioning for a general block containing a call to $f$ by using an approach similar to the
one for letrec blocks. In the case of a function call, the information of interest is the possible input/output dependencies. If those are unknown, we need only produce a partitioning that is correct for all possible such dependencies. In the same way as cycles are lifted out for a letrec block, the potential dependencies in a general block containing a call to $f$ can be lifted out. In this case, lifting out dependencies corresponds to removing the function call and introducing as additional inputs the outputs to the function, and as additional outputs the inputs to the function. For example, to find a correct partitioning for

$$h(w) = \{ x = w * 2;$$

$$y = f(x);$$

$$r = y + x$$

$$\text{in } r \}$$

we can use the information from analyzing

$$h(w, y) = \{ x = w * 2;$$

$$r = y + x$$

$$\text{in } r, x \}$$

By applying the partitioning analysis indirectly, once again correct but non-optimal partitions are generated. The partitions are correct for any possible dependency of an output on an input, rather than only for the specific dependencies arising from the function call. Continuing with the example above, the partition will be correct whether $y$ depends on $x$ or not. Better partitions can be produced if $f$ is known, and in subsequent parts we will show how information about a function can be used in the case of a function call, as well as in the analysis of recursive functions.

### 7.3.2 Higher-order procedures and data structures

Our algorithm is limited to a first order, functional language with simple data types. It can therefore be extended to include data structures, higher-order procedures, as well as recursive data structures. Also, rather than requiring that the source code
handed to our algorithm be in the form of letrec blocks, the algorithm could be extended to handle letrec blocks directly.
Appendix A

Upperbound on Number of Iterations

In this appendix, we present a proof that in the demand/tolerance algorithm, demand, dependence, and tolerance labels can be merged in at most two iterations. From this proof we deduce that the equivalent iterations in our algorithm also have an upperbound of two iterations.

In order to prove the upperbound, we try to construct a graph that leads to more than two iterations and show that this leads to a contradiction. Rather than merging nodes with the same demand and tolerance sets, let us first consider merging nodes with the same dependence and demand labels. In the latter case, one would have to be in one of the two following scenarios in order to need more than two iterations.

Scenario 1:

1. Merge nodes with the same demand sets: Unite nodes that have the same demand and same dependence sets or no nodes can be united on this step.

2. Merge nodes with the same dependence sets: This step must involve uniting at least two nodes, say $n_1$ and $n_2$, that have the same dependence sets but different demand sets (such that one is not a subset of the other) into node $n'$.

3. Merge nodes with the same demand sets: This step must involve using the demand set obtained in step (2), to unite a node, say $n_3$, that has the same
demand set as \( n' \) but a different dependence set than \( n' \) (such that one is not a subset of the other) and obtain a node \( n'' \).

4. Merge nodes with the same dependence sets: This step must involve using the new dependence set obtained in step (3), to unite a node, say \( n_4 \), that has the same dependence set as \( n'' \) but a different demand set (such that one is not a subset of the other).

5. Merge nodes with the same demand sets: With the new demand set obtained in step (4), unite a node, say \( n_5 \), that has the same demand as \( n'' \) and a different dependence set.

Scenario 2:

1. Merge nodes with the same demand sets: This step must involve uniting at least two nodes, say \( n_1 \) and \( n_2 \), that have the same demand set but different dependence sets (such that one is not a subset of the other) to obtain a node \( n' \).

2. Merge nodes with the same dependence sets: This step must involve using the new dependence set obtained in step (1), to unite a node, say \( n_3 \), that has the same dependence set as \( n' \) but a different demand set (such that one is not a subset of the other), and obtain node \( n'' \).

3. Merge nodes with the same demand sets: This step must involve using the new demand set obtained in step (2), to unite a node, say \( n_4 \), that has the same demand set as \( n'' \) but a different dependence set (such that one is not a subset of the other), and obtain node \( n''' \).

4. Merge nodes with the same dependence sets: This step must involve using the new dependence set obtained in step (3), to unite a node, say \( n_5 \), that has the same dependence set as \( n''' \) but a different demand set (such that one is not a subset of the other), and obtain a node \( n'''' \).
5. Merge nodes with the same demand: With the new demand set obtained in step (4), unite a node, say \( n_6 \), that has the same demand set as \( n''' \) and a different dependence set.

If one were to start by uniting nodes with the same dependence sets, (Scenario 1) reduces straightforwardly to a scenario that only takes two iterations. Step (1) would happen while nodes with the same dependence sets in step (2) are merged, so step (1) and (2) would just collapse into one step. Then, (Scenario 1) becomes the same as (Scenario 2) in the case where we start by merging nodes with the same dependence set. So let us leave (Scenario 1) aside for a moment, and look at (Scenario 2). Then by symmetry of demand and dependence sets, from (Scenario 2) we will be able to draw conclusions about (Scenario 1).

Let us start our analysis of (Scenario 2) by trying to construct a graph that behaves this way with dependence/demand merging.

1. \( n_1 \) and \( n_2 \) must have the same demand sets but different dependence sets such that one is not a subset of the other, so: \( n_1 \) and \( n_2 \) must be on different paths that lead to the same set of outputs \( O \). If \( n_1 \) and \( n_2 \) were on the same path (assuming \( n_1 \) is an ancestor of \( n_2 \)), then the dependence set of \( n_2 \) would be a subset of the dependence set of \( n_1 \) (straight from recursive definition of dependence sets) which is not allowed. Furthermore, \( n_1 \) and \( n_2 \) need to be descendants of different set of input nodes \( I \). Otherwise, \( n_1 \) and \( n_2 \) would have the same dependence set.

2. \( n_3 \) must have the same dependence set as \( n' \) but a different demand set such that one is not a subset of the other: \( n_3 \) cannot have the same demand set as \( n_1 \) and \( n_2 \), otherwise all nodes would have been merged in step (1). Furthermore, \( n_3 \) cannot have a demand set which is a subset of \( n_1 \) or \( n_2 \), so \( n_1 \) cannot be a descendant of \( n_1 \) or \( n_2 \) (straight from recursive definition of demand sets). The other restriction on \( n_3 \) is having the same dependence set as \( n' \), that is, the node obtained by merging \( n_1 \) and \( n_2 \). Therefore, \( n_3 \) must be a child of an ancestor of \( n_1 \) and an ancestor of \( n_2 \) (it does not have to be a common ancestor of \( n_1 \) and
and have a path to an output node that is not in $O$.

3. $n_4$ must have the same demand set as $n''$ but a different dependence set such that one is not a subset of the other. $n_4$ cannot have the same demand set as $n_1$ or $n_2$ otherwise those nodes would have been combined in (1). $n_4$ cannot have the same demand set as $n_3$ otherwise those nodes would have been combined in (1). $n_4$ must therefore be an ancestor of $n_1$ and $n_2$ and $n_3$, so that $n_4$ has a different demand set than $n'$ and $n_3$, but the same as $n''$. $n_4$ could also be an ancestor of $n_1$, $n_2$ and $n_3$’s children. Though is these children have the same demand set as their parents, they would have been combined with their parents in step (1) which is allowed. Either way this makes $n_4$ an ancestor of $n''$. However, $n_4$ cannot be an ancestor of $n''$, because this means that the dependence set of $n_4$ is a subset of $n''$ which is not allowed. So it is impossible to construct a node $n_4$ to obtain step (3). All we can do is construct a node $n_4$ that has the same demand set as $n''$ and the same dependence set as $n''$ so there are only two possible steps in this scenario. Node $n_4$ would get combined with node $n_3$ and $n'$ in step (2).

Since only two steps are possible in (Scenario 2), this means that only one iteration through the algorithm is needed in this scenario. Therefore, by symmetry we can deduce that in (Scenario 1) after step (1) only two steps can occur, which leads to at most two iterations. However, as observed earlier step (1) of (Scenario 1) is extraneous since if we were to start by merging nodes with the same dependence sets, step (1) and (2) would collapse into the same step.

One could modify the algorithm to check whether the only nodes that can be merged in step (1) of the algorithm have same demand and dependence sets. If they do, then start merging with same demand sets and then merge with same dependence sets. Otherwise, start merging those with same dependence sets. Then, we obtain an algorithm that requires only one iteration.

Merging with tolerance subsumes merging with dependence: If two nodes have the same dependence set then they also have the same tolerance set. Therefore, from our
observations on dependence/demand merging, we can conclude that there is an upper bound of one iteration needed to merge nodes in the demand/tolerance algorithm. Since using predicated set labels reduces to using sets for basic blocks corresponding to each possible control flow of a general block, the upperbound also holds for our algorithm.
Bibliography

[1] Arvind, Jan-Willem Maessen, R. S. Nikhil, and J. Stoy. $\lambda_S$: An implicitly parallel $\lambda$-calculus with letrec, synchronization and side-effects. (in preparation).


