Partitioning Non-strict Languages for Multi-threaded Code Generation

by

Satyan R. Coorg

B.Tech, Computer Science and Engineering, Indian Institute of Technology, Madras

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1994

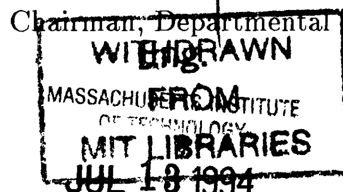
© Satyan R. Coorg 1994

The author hereby grants to MIT permission to reproduce and to
distribute copies of this thesis document in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 10, 1994

Certified by ______________________________
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________________
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students



Partitioning Non-strict Languages for Multi-threaded Code Generation

by

Satyan R. Coorg

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 1994
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

In a *non-strict* language, functions may return values before their arguments are available, and data structures may be defined before all their components are defined. Compiling such languages to conventional hardware is not straightforward; instructions do not have a fixed compile time ordering. Such an ordering is necessary to execute programs efficiently on current microprocessors. *Partitioning* is the process of compiling a non-strict program into *threads* (i.e., a sequence of instructions). This process involves detecting data dependencies at compile time and using these dependencies to "sequentialize" parts of the program.

Previous work on partitioning did not propagate dependence information across recursive procedure boundaries. Using a representation known as *Paths* we are able to represent dependence information of recursive functions. Also, we incorporate them into a known partitioning algorithm. However, this algorithm fails to make use of all the information contained in paths. To improve the performance of the partitioning algorithm, we design a new algorithm directly based on paths. We prove the correctness of the new algorithm with respect to the operational and denotational semantics of the language. Finally, we suggest extensions to the algorithm to handle data-structures and to make use of context information during partitioning.

Thesis Supervisor: Arvind

Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I am very grateful to my thesis advisor, Arvind, for his guidance, encouragement, and patience. I also thank him for helping me to finish on time.

I thank all the members of the Computation Structures Group for their continuing help and support. I thank Shail Aditya for carefully reading drafts of my thesis and helping me present my ideas in an organized manner. I thank Boon Ang for the discussions we had on partitioning. Yuli Zhou helped me get started on the abstract interpretation path by giving pointers to the literature in the field.

I am very grateful to my parents for their love and affection. I also thank my friends and relatives back in India for the interest they have shown in me and for their support.

To my friend,

K. G. Venkatasubramanian (1970-1993)

Contents

1 Introduction **10**
- 1.1 Non-strictness in Functional Languages 11
  - 1.1.1 Lazy Evaluation 12
  - 1.1.2 Lenience: Non-strictness without Laziness 13
  - 1.1.3 The Problem 13
- 1.2 Id and its Compilation 14
- 1.3 Background 17
  - 1.3.1 Strictness Analysis 17
  - 1.3.2 Dependence Analysis 20
  - 1.3.3 DD Partitioning 21

2 SP-TAC: Syntax and Semantics **28**
- 2.1 Abstract syntax of the language 29
- 2.2 Notation 29
- 2.3 Operational Semantics 31
  - 2.3.1 Canonical Representation of Terms 31
  - 2.3.2 Rewrite Rules of SP-TAC 32
  - 2.3.3 Observable results 33
- 2.4 Denotational Semantics 34

3 Analyzing Recursion using Paths **36**
- 3.1 Path Semantics 36
- 3.2 Abstracting Paths 45
- 3.3 Integrating Paths and DD 50
- 3.4 Summary 54

4 Partitioning: A New Approach **55**
- 4.1 Partitioning as transformation 55
  - 4.1.1 Correctness of a Partitioning 59
  - 4.1.2 Summary 60
- 4.2 Partitioning: Preliminary Algorithms 61
  - 4.2.1 Subpartitioning 61
  - 4.2.2 Converting cyclic to acyclic blocks 63
  - 4.2.3 Summary 66
- 4.3 Partitioning strict blocks 66
  - 4.3.1 Dependence graph based Correctness of a partitioning 68
  - 4.3.2 Safety of the Dependence graph Criterion 69
  - 4.3.3 Strict blocks: Correctness of the AB algorithm 70

4.3.4 Summary 73
4.4 Partitioning non-strict blocks 73
4.4.1 N-block based Correctness of a Partitioning 80
4.4.2 Correctness proof of the AB algorithm 84
4.4.3 Summary 88
4.5 Appendix: Complexity of Partitioning 89

5 Extensions **92**
5.1 Partitioning with Contexts 92
5.2 Data Structures 97
5.2.1 Abstract Paths for Data-structures 98
5.2.2 Partitioning with data-structures 99
5.3 Side Effects 102
5.4 Summary 103

6 Conclusion **104**
6.1 Further Research 104
6.1.1 Implementation 104
6.1.2 Higher Order Functions 105
6.1.3 Recursive Data Types 105
6.1.4 Subscript Analysis 105
6.1.5 Efficiency or Accuracy 106

List of Figures

1.1 Structure of an Id Compiler 15
1.2 Dependence and Separation Graphs 21
1.3 Dataflow Graph Operators 22
1.4 Iterated Partitioning 27

3.1 Different approaches to Partitioning 37
3.2 Structure of the Path Domain 38
3.3 Example of Paths used for DD 52

4.1 A Dependence Graph 62
4.2 B stage of algorithm AB 68
4.3 Proof of $Prop(r)$ for A sets 86
4.4 Proof of NP-hardness 89

Chapter 1

Introduction

Functional programming languages can be divided into two classes *strict* and *non-strict.* In a *non-strict* language, functions may return values before their arguments are available, and data structures may be defined before all their components are defined. Such languages give greater expressive power to the programmer than a strict language. The programmer does not need to worry about carrying dependencies across function boundaries; non-strictness enable these dependencies to be satisfied at run time.

Compiling such languages to conventional hardware is not straightforward. Non-strictness gives the programmer the ability to feed results of a function back into its arguments. We cannot "wait" for all the arguments and then call the function – this would cause a deadlock in the program. Thus, instructions in a non-strict language do not have a fixed compile time ordering. The instructions may execute in any order satisfying the data dependencies.

However, given that the most popular languages like C, Fortran are sequential (i.e., there is a definite ordering of instructions), processors which execute these languages efficiently are highly tuned to exploit this ordering of instructions. Thus, such an ordering is necessary to execute programs efficiently on current microprocessors. *Partitioning* is the process of compiling a non-strict program into *threads* (i.e., a sequence of instructions). This process involves detecting dependencies at compile time and using these dependencies to "sequentialize" parts of the program.

In the rest of the chapter, we deal with the following issues. First we discuss non-strictness and the need for a partitioning algorithm. Then, we give a brief overview of a non-strict language Id [27] and its compilation process. We end the chapter by discussing

the state of the art algorithms for partitioning (and other related topics).

Chapter 2 defines the language framework employed in the thesis and provides operational and denotational semantics for this language. In Chapter 3 we introduce *paths*, a way of representing dependence information and apply it to previous partitioning algorithms. Chapter 4 is the main contribution of this thesis. It defines a new approach to partitioning, viewing it as a source to source transformation. We also present a new algorithm for partitioning and prove its correctness using operational and denotational semantics. Chapter 5 extends the basic partitioning algorithm in Chapter 4 in several directions. Chapter 6 concludes.

1.1 Non-strictness in Functional Languages

In this section, we focus on non-strictness in languages – the property that arguments are passed unevaluated to procedures and data structures. We show how non-strictness allows fuller use of recursion than is possible under strict evaluation. Example 1.1 illustrates non-strictness in data-structures and example 1.2 illustrates non-strictness arising from the conditional statement. These examples are taken from [35].

Example 1.1:

```
{a = mk_tuple2(2,b);
 b = select1(a)
 in
   a}
```

Example 1.2:

```
f(n)={p = eq? n 0;
      a = if p then { in 3} else { in d};
      b = if p then { in c} else { in 4};
      c = a + 2;
      d = b + 1;
      e = c * d
      in
        e}
```

The above examples are given in terms of a recursive (*letrec*) block. There is no ordering of the statements present in the block. **mk_tuple** and **select** operators make a "tuple" of values and select any component of a tuple respectively.

In a *strict* evaluation, where all expressions are strict in the variables they contain, both the examples produce a *deadlock*. In example 1.1, the tuple corresponding to the variable

a cannot be returned until b is defined. However, b cannot be defined until the tuple a is defined – leading to a deadlock. A similar reasoning shows that example 1.2 also produces a deadlock under strict evaluation. In strict languages like Scheme [32], the *letrec* is not used to define such "circular" dependencies, but used to define *recursive functions* .

Now consider a *non-strict* evaluation of the examples. In example 1.1 the tuple a is returned with its first component containing 2 and the second component undefined. Selecting the first component of this tuple gives the value 2 which in turn makes the variable b defined. Thus, the whole program returns the tuple $(2, 2)$. Now, consider example 1.2. Suppose that the value of the predicate is *true*. Then, a will get the value 3. This means that c will get the value 5 and b in turn gets the value 5. The final value returned from the block is $5*6 = 30$. The case when the predicate evaluates to $false$ is similar. The important point to note is that the order of evaluation of variables will be different in that case – b will get a value before a does. The examples can be easily modified to involve non-strictness of functions (by abstracting the non-strict expressions into separate functions).

We now consider two ways of implementing this non-strictness: lazy evaluation and lenient evaluation.

1.1.1 Lazy Evaluation

In the examples given to illustrate non-strictness, evaluation of some expressions needed to be delayed until their evaluation was possible – strict evaluation would try to evaluate them too early, leading to a deadlock. Lazy evaluation, the evaluation strategy followed by languages like Haskell [17], delays expressions until the last possible moment, i.e., when the execution cannot proceed without it. All the examples given produce correct answer under lazy evaluation, since lazy evaluation will certainly introduce sufficient delays to insure that the delayed expressions are executable. In fact, if there is any evaluation order which will produce an answer from a program, lazy evaluation will also produce an answer to that program [42].

Laziness grants an additional flexibility to the programmer: the ability to create and manipulate infinite data structures. A well known example in literature is the sieve of Eratosthenes for finding prime numbers [15]. The program works by defining an infinite list of integers, and using this list to select out the primes. Even though the list is infinite, lazy evaluation scheme actually builds the list long enough to return the nth prime – no more.

A strategy without laziness would build the infinite list and would fail to terminate.

1.1.2 Lenience: Non-strictness without Laziness

Lenient evaluation is defined in [35] to be an evaluation strategy which achieves non-strictness but not necessarily laziness. For the implementation, this implies that expressions must be delayed until their input data are available, but not necessarily any longer. For the programmer, it means unrestricted use of letrec and flexibility in data dependencies, but not consumer-directed control structure required for the finite use of infinite data structures. It is shown in [35] that programs can be implemented more efficiently in the lenient strategy than possible in the lazy strategy. In this thesis, we will be primarily concerned with non-strictness arising from the lenient strategy. In other words, we will develop compilation strategies to ensure that a program producing a correct answer using the lenient strategy does not deadlock after it is compiled.

1.1.3 The Problem

Consider the following program written in Id [27], a non-strict language with a functional core.

```
f(a,b)={c = a + 1;
        d = b * 2;
        in
          (c,d)}
```

Now, it is not safe in general to compile the above function into a single thread for the reason that there could be a dynamic dependence between the two instructions outside the function body. For example, the following context creates a dependence from the * instruction to the + instruction. Non-strictness of the function f ensures correct results even though a result is fed back to an argument.

```
{(x,y)=f(y,2)
 in
   y}
```

We can easily construct another context in which there is a dependence from the + instruction to the * instruction. Thus, there is no fixed ordering which can be determined by the two instructions.

Informally, the problem is to produces a set of threads given a program in Id. Each thread is a subset of instructions of a procedure body, which satisfies: [37]

1. A compile-time instruction ordering can be determined for the thread which is valid for all contexts in which the procedure is invoked.

2. Once the first instruction in a thread is executed, it is always possible to execute each of the remaining instructions without pause, interruption, or execution of instructions from other threads.

Threads of this form are required to implement a language on parallel multithreaded hardware (e.g., Monsoon [30, 29], and *T [28, 3]). They also yield efficient implementations on conventional architectures when combined with a suitable abstract machine such as TAM [13]. In a later chapter, we formalize this notion of a thread and cast it into an operational semantic framework.

1.2 Id and its Compilation

Our thesis is targeted toward producing a partitioning algorithm for use in a compiler for the Id language. Id [27] is a functional language extended with I-structures [5] and M-structures [8]. Like any other modern functional language, Id has the standard features including higher order functions, algebraic data types, pattern matching and a Hindley-Milner based type inference algorithm. In this thesis, we concentrate on a first order subset of Id allowing I-structures as the only means of achieving side effects.

We now give an overview of an Id compiler being implemented in Id [46] so that the reader can put the partitioning stage of the compiler in this perspective. The compiler reads an Id source file and produces Multi-threaded machine code to run on conventional microprocessors (Figure 1.1). In this process, it uses five intermediate representations AID, KID, KID Graphs (KG), P-TAC, and Partitioned P-TAC.

AID is very close to Id in terms of its syntactic constructs. KID (for Kernel Id) is a much smaller language: syntactic sugars in AID are removed, complex pattern matching is replaced by simple multi-way branches, and expressions are flattened by introducing temporary variables for each intermediate result. KG is a graph representation of KID. P-TAC's main difference from KG is that data structure and closure handling are exposed

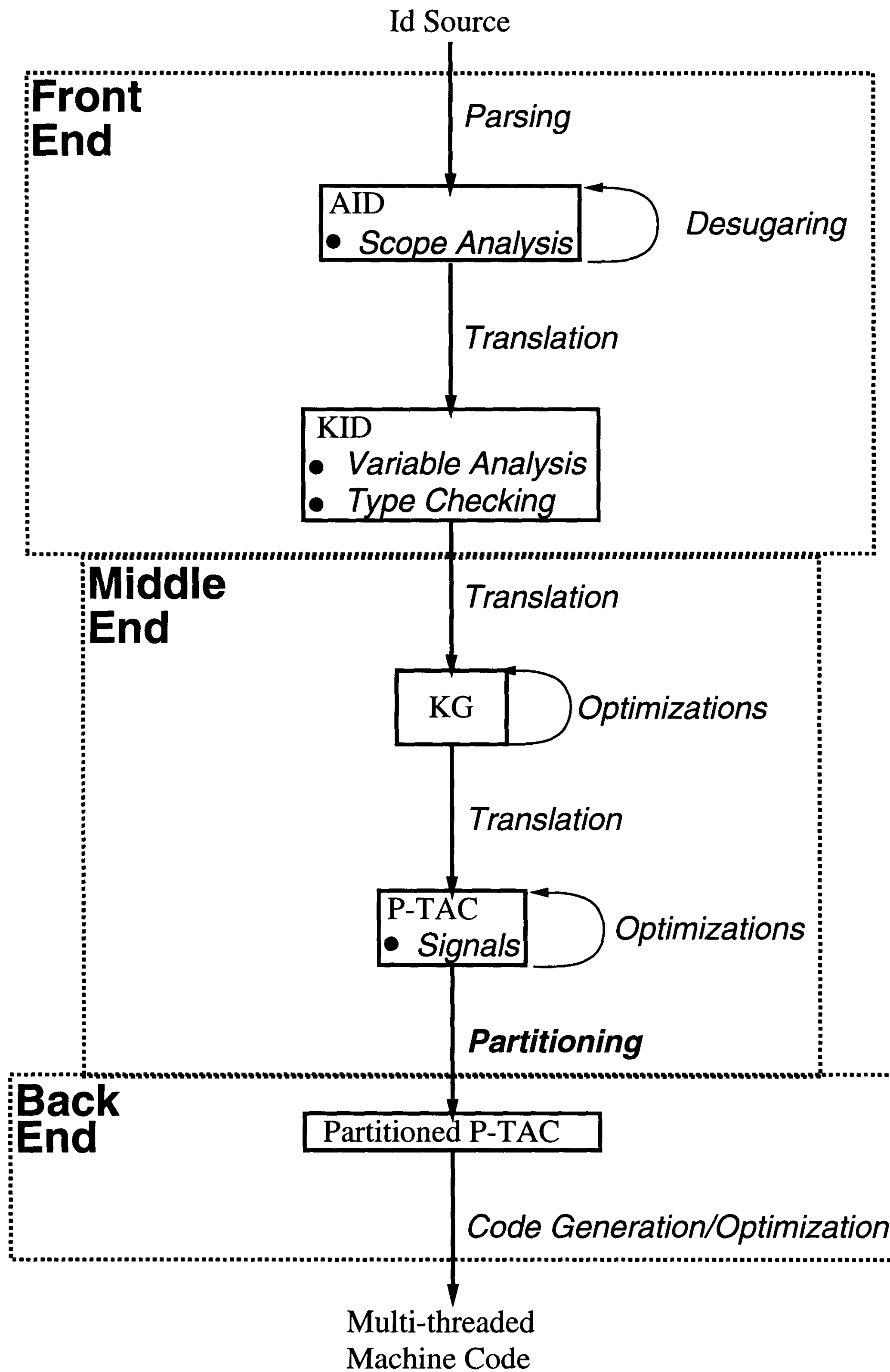


Figure 1.1: Structure of an Id Compiler

in terms of primitive operations on the heap. Partitioned P-TAC groups together nodes in P-TAC which belong to the same partition.

The modules of the compiler are organized in the conventional arrangement of a *front end*, *middle end* and a *back end.* A short description of the functions performed by each module is given below: (we omit descriptions of translation modules, whose functionality is obvious)

1. **Parsing**: This is modified version of Berkeley YACC to generate a LALR[1] parser in Id.

2. **Desugaring**: This is a macro expansion phase in which multi-clause definitions and functions, list and array comprehensions are replaced by simpler AID constructs.

3. **Scope Analysis:** This relates occurrences of variables to their definition.

4. **Variable Analysis**: Annotates each KID subtree with the set of variables that occur free in it.

5. **Type Checking**: It processes type annotations, resolves overloading and infers types for each subexpression in the KID tree.

6. **Optimizations on KG**: Performs *constant propagation*, *common subexpression elimination*, *fetch elimination* and *call substitution.*

7. **Optimizations on P-TAC**: Same optimizations as in the KID stage are performed on P-TAC graphs. A lot of redundancy is generated in the translation from KID to P-TAC and these optimizations try to eliminate them.

8. **Signals**: *Signals* are added to the P-TAC graph to ensure proper sequencing of barriers, and for use in reclaiming storage allocated for function calls.

9. **Partitioning**: This generates a set of threads (a sequence of P-TAC operations) given the P-TAC graph of the program. Developing algorithms for this module is the goal of the thesis. The algorithms presented in this thesis are yet to be implemented.

10. **Code Generation/Optimization**: This is the final stage of the compiler producing executable machine code. The code is *multi-threaded* for two reasons: one, we would like the code to run in *parallel*, two, to ensure that non-strictness of the language is guaranteed by the implementation.

1.3 Background

In this section, we give a brief overview of some related work arising out of compiling *lazy* languages like Haskell. We also discuss previous approaches to partitioning.

In lazy languages, an expression is evaluated only when it is required for an answer to be produced by the program. This raises problems in generating efficient sequential code, as the order of expression evaluation cannot be determined at compile time. A lot of research has been done in the lazy language community to find good solutions to this problem. We will discuss *strictness analysis* in this section, as it is particularly relevant to the ideas presented in this thesis.

The seminal work in [35] defined the problem of partitioning and identified the precise reasons due to which this compilation step is required. This work was based on firm theoretical foundation and used dependence analysis to identify potential and certain dependencies in programs. The actual partitions were obtained by a graph coloring technique which converted programs into threads while satisfying the detected dependencies. This is a global approach; we need information about the whole program to apply this technique. We call this approach the *Dependence Analysis* approach.

Another less theoretical, more pragmatic approach (called Demand and Dependence (DD) partitioning) started in [21], progressed through [36, 16, 38] and culminated in [37]. The approach began as a local approach. That is, the initial algorithms worked as follows: look at "small" regions in programs and see whether they can be sequentialized. This "local" analysis was extended to propagate local information to the entire program in [37].

1.3.1 Strictness Analysis

As we have discussed earlier, lazy evaluation requires that argument expressions be passed to procedures unevaluated, and that primitives like arithmetic operators must cause such unevaluated expressions to be evaluated. Techniques which have emerged for lazy evaluation are Turner's combinators [41], Henderson's force-delay transformation [15], and Graph Reduction [45, 22, 31]. One thing common to these implementations is that they can produce more efficient code if they can evaluate an argument directly, instead of passing an unevaluated expression. However, given the constraints of lazy evaluation, this is not safe in general. *Strictness Analysis* determines when an argument can be safely evaluated (i.e.,

without causing a program to diverge).

Strictness analysis is based on the notion of a strict function:

Definition 1.1 *A function f of n arguments is* strict *in its ith argument if*

$$f(x_1, \ldots, x_{i-1}, \perp, x_{i+1}, \ldots, x_n) = \perp$$

for all values of $x_1, \ldots, x_{i-1}$ and $x_{i+1}, \ldots, x_n$.

As Mycroft points out [26], we can safely evaluate an argument to a function f which is strict in that argument. If f terminates, the argument would have been evaluated, as the argument is needed to produce a value for the result. If f does not terminate, nothing is changed by evaluating the argument. The definition for strictness given above gets more complicated when extended for data-structures and higher order functions.

Strictness analysis attempts to determine in which arguments, if any, the functions of a program are strict. As strictness is a undecidable property, the analysis has to make approximations at compile time. Many approaches to strictness analysis have been developed including *abstract interpretation* [26, 10], *backwards analysis* [19], and *type inference* [25]. A good overview of the field is given in [2].

We give a brief overview of the abstract interpretation approach and point out its ability in dealing with recursive programs. In this approach, the idea is to *execute* the programs with partial information about the inputs and deduce partial information about the outputs. Instead of the inputs having the run time values, then can have two *abstract values* **1** or **0**. **1** denotes any possible value and **0** denotes *non-termination* (i.e., the value is undefined). For each function f we construct an *abstract function* $f^{\#}$ which takes abstract values as its inputs and returns abstract values as its outputs. For example, the abstract functions for the + operator and the conditional (**if**) are given below:

$$\begin{aligned} +^{\#}(x^{\#}, y^{\#}) &= x^{\#} \wedge y^{\#} \\ \mathbf{if}^{\#}(x^{\#}, y^{\#}, z^{\#}) &= x^{\#} \wedge (y^{\#} \vee z^{\#}) \end{aligned}$$

The operators $\wedge$ and $\vee$ are similar to the usual **and** and **or** operators on boolean values. The $+^{\#}$ abstract function mirrors the fact that + needs both the arguments to return a defined value; if one of the arguments is undefined, the result is undefined. The **if** operator is more interesting. When is its result defined? Clearly, the predicate ($x^{\#}$) must be defined.

We know that one of the values $y^\#$ or $z^\#$ must be defined, but at compile time we do not know which. Thus, the analysis makes a (conservative) approximation that if one of the branches is defined, the result of the **if** is defined.

Using the above method, we can know whether a function f is strict in its ith argument by evaluating the expression $f^\#(\mathbf{1},\ldots,\mathbf{1},\mathbf{0},\mathbf{1},\ldots,\mathbf{1})$. Except the ith argument, the arguments to $f^\#$ are all $\mathbf{1}$. If this function returns $\mathbf{0}$, we *know* that the function f is strict. If it returns $\mathbf{1}$, the function may or may not be strict. However, a straightforward evaluation of $f^\#(\mathbf{1},\ldots,\mathbf{1},\mathbf{0},\mathbf{1},\ldots,\mathbf{1})$, when f is recursive creates some problems. Consider the following function and its abstraction:

$$
\begin{aligned}
f(x,y) &= \mathbf{if}((y == 0), x, f(x, y-1)) \\
f^\#(x^\#, y^\#) &= \mathbf{if}^\#((y^\# \wedge \mathbf{1}), x^\#, f(x^\#, y^\# \wedge \mathbf{1}))
\end{aligned}
$$

The abstract function can be simplified to:

$$
f^\#(x^\#, y^\#) = y^\# \wedge (x^\# \vee f^\#(x^\#, y^\#))
$$

Suppose we want to determine whether f is strict in x. We should evaluate $f^\#(\mathbf{0},\mathbf{1})$. However, we need to know the value of $f^\#(\mathbf{0},\mathbf{1})$ before (as it occurs on the right hand side of the function definition). Trying to do a straightforward evaluation would result in the analyzer itself not terminating. To avoid this, the "circularity" is resolved by using fix point iteration. To start out, we *assume* that the function is strict in all its arguments (i.e., $f^\#(\ldots)$ returns $\mathbf{0}$ for all inputs). We get,

$$
f^\#(\mathbf{0},\mathbf{1}) = \mathbf{1} \wedge (\mathbf{0} \vee \mathbf{0}) = \mathbf{1} \wedge \mathbf{0} = \mathbf{0}
$$

It is easily verified that this is indeed the value of the function in its limit, i.e., f is strict in x.

To check for termination of our fixed point iteration, we need to check for equality of abstract functions obtained from successive iterations. If we assume a naive implementation of the function as a table with 2^n entires (n is the number of arguments to the function which returns a single value), each iteration will take exponential time. A lot of work has been done in choosing a "good" representation for the function so that fix point iteration can be speeded up. A good example is the *frontiers* representation [11].

Finally, we note that strictness analysis has been extended to analyze higher-order functions [10], data structures [43, 44], polymorphic functions [1, 20, 40, 6].

1.3.2 Dependence Analysis

In this section, we give a brief overview of the Dependence Analysis approach as presented in [35]. The algorithm produces "suspendable" threads given a non-strict program as it inputs. Threads are suspendable in the sense that if a value they need is unavailable, they are suspended. They are resumed later when the value becomes available. Note that this notion of a thread is different from the one given previously.

The algorithm performs an approximate dependence analysis to construct a dependence graph of a block. The dependence graph reflects all possible dependencies in the graph. Dependencies are of two types: *certain*, which we know at compile time and *potential* which we do not. Using the dependence graph we can compute a *separation graph* which has edges between two variables a and b if there is a *potential dependence path* from a to b and vice versa. A potential dependence path is simply a path in the dependence graph with at least one potential dependence edge. The partitions are obtained by a coloring of the separation graph, which implies that variables which have to be separated are put in different threads. An example of this approach is given below.

Figure 1.2 shows the dependence and separation graph of example 1.2. Solid edges denote certain dependence and dashed edges denote potential dependence. We have already seen that variables a and b cannot be put in the same thread (as the order in which they get their values is not fixed at compile time). This constraint shows up in the separation graph as an edge between the variables a and b. By a coloring of the separation graph we can obtain the partitions $\{n, p, a, c, e\}$ and $\{b, d\}$.

Problems: The algorithm has to put a potential dependence edge between any result of a function and any of its inputs. This is due to the fact that there could be arbitrary feedback of the results in the language. This gives rise to many potential dependence paths and many separation constraints, producing small threads. Some heuristics were suggested in [35] to overcome this limitation.

1. If a function is *strict* in an argument, the feedback edge is not added.

2. Potential edges are eliminated whenever they conflict with a certain dependence path in the dependence graph.

3. Given a set of vertices in the graph which form a *strict region*, the strict region is collapsed into a single vertex. A strict region is a set of vertices in the dependence

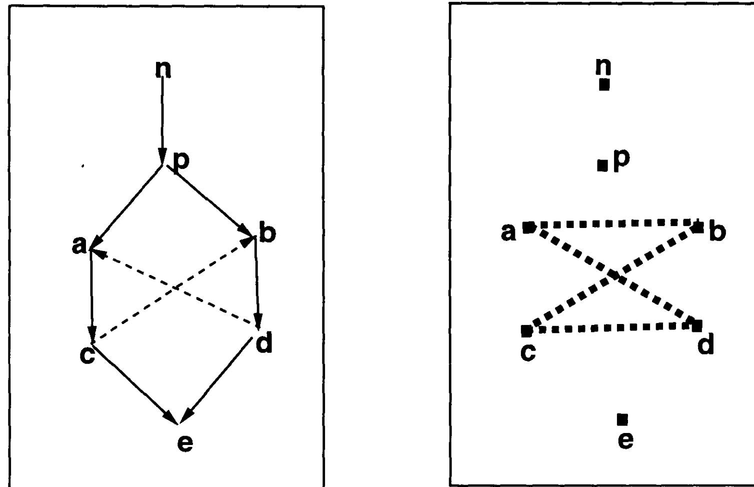


Figure 1.2: Dependence and Separation Graphs

graph where all dependencies in the region are strict and there is only one (output) vertex which depends on these vertices.

The later partitioning algorithms expanded this notion of a strict region to produce better partitions.

1.3.3 DD Partitioning

We only give a brief overview of the approach, the reader is encouraged to read [37] for more details. The goal of this approach is to generate a set of threads from a given program. The algorithms given are greedy, and seek to maximize the size of the threads as much as possible. The algorithm works on programs represented as *dataflow graphs* which are described below.

Dataflow Graphs: Programs to be partitioned are expressed in a *structured dataflow graph*, an intermediate form into which Id is compiled [33]. A structured dataflow graph consists of a collection of acyclic graphs describing *basic blocks* and *interfaces*. A basic block corresponds to a group of operators with the same control dependence. For example, operators comprising the "then" arm of the conditional, excluding those in nested conditionals, are a basic block. **Basic Blocks:** These are directed acyclic graphs with the vertices representing operators, and the edges representing flow of values. Some of the operators used to compile Id are shown in figure 1.3. These consist of the usual primitive operators (like

$+, -$, etc.), operators for sending a value to a function and receiving a value returned from a function. The send and receive operators perform the procedure linkage operations. For data structures, there are operators to read *(I-fetch, fetch)* and operators to write *(I-store, store)*.

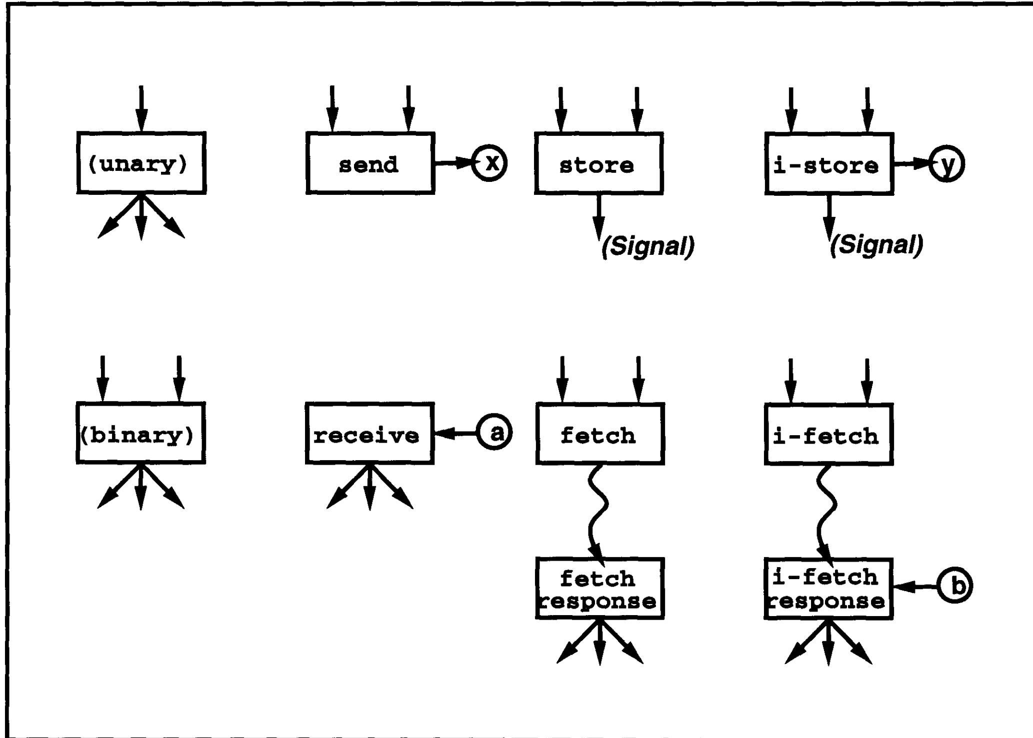


Figure 1.3: Dataflow Graph Operators

An edge in the graph indicates the flow of an operand; it also indicates a *certain* dependence between two operands. A certain dependence is one that can be detected at compile time. The dataflow graphs used in the algorithm have *squiggly* edges, indicating a split-phase operation (i.e., one where the processor does not "wait" for the operation to be completed (e.g. I-fetch)). The partitioning algorithm must "break" a thread at each squiggly edge to implement the split-phase transaction.

There is one other kind of dependence namely, *potential* dependence. This is possible dependence which cannot be completely determined at compile time. These dependencies are represented using sets of labels called inlets and outlets. An inlet (outlet) is a name for a potential incoming (outgoing) dependence. Note that we have introduced inlet annotations in receive and i-fetch operators as we do not know (in general) which nodes these vertices

depend upon.

A vertex with no inlets does not have any (potential) dependence from any other vertex. Though the dependencies of vertices with non-empty inlet/outlet annotations are not known, the algorithm uses the inlets and outlets to find "equivalence" between various vertices, and uses these equivalences to generate partitions.

Interfaces: Interfaces are mechanisms for a basic block "calling" another. Each basic block with n input vertices and m output vertices is basically an n-argument, m-result function. Communication between basic blocks is accomplished through send and receive vertices. On the callers side, there is a send for each argument to be sent to the callee and a receive for each result returned by the callee. The callee uses receives to obtain arguments from the caller and sends to send the results back. For a function, there is a single place where it is defined (the *def* place) and used in other basic blocks (the *use* sites). Thus, a function typically has a single-def multiple-use interface. The algorithm treats the conditional as a generalization of the procedure call: there are two defined functions (the "then" and "else" basic blocks) and a single call (depending upon the value of the predicate). Thus, a conditional is a single-use multiple-def interface.

Partitioning Basic Blocks

The algorithm uses demand/dependence sets to do partitioning. At each stage it seeks to put nodes with the same demand (or dependence) sets in the same partition.

Definition 1.2 (Dependence Sets) *A Dependence set of a node is a set of inlets on which it depends [21]:*

$$Dep(v) = \bigcup_{u \in Pred^*(v)} Inlet(u)$$

where Inlet(u) is the set of inlet names that annotate u, and $Pred^*(v)$ is the set of nodes from which there is a path to v.

Definition 1.3 (Demand Sets) *A Demand set of a node is a set of outlets which depend on it [38]:*

$$Dem(v) = \bigcup_{u \in Succ^*(u)} Outlet(u)$$

where Outlet(u) is the set of outlet names that annotate u and $Succ^*(v)$ is the set of nodes to which there is a path from v.

The Demand stage of the algorithm works as follows:

1. Compute the demand sets of each node. As basic blocks are acyclic this can be done by a *backward* pass.

2. Partition the graph into (maximal) sets of nodes with the same demand sets.

The dependence stage of the algorithm is analogous.

The partitions obtained by one of the two stages can have two nodes connected by a *squiggly* arc, violating the second condition for a thread. Thus, we need to *subpartition* the partitions we have obtained. The (backward) subpartitioning algorithm works as follows:

1. For each node v in a thread, compute the maximum distance ($Subpart(v)$) (with squiggly edges having weight 1 and other edges having weight 0) to the "leaves" of the thread.

2. Form subpartitions of the thread putting nodes with identical $Subpart()$ together.

An analogous forward subpartitioning algorithm can be defined. Observe that the subpartitioning algorithms cannot introduce a cycle in the thread, as subpartition numbers are monotonically decreasing (or increasing) along a path.

A stage in the DD algorithm is to perform Demand (or Dependence) partitioning with subpartitioning. The informal argument given in [37] to prove correctness of the algorithm uses the fact that any static or dynamic path between two nodes in the same thread cannot contain nodes in some other thread.

1. No static path can exist due to the fact that the dependence (or demand) sets are monotonically increasing along the path.

2. Assume a dynamic path exists between two nodes in the same thread. Then, it must be completed through one of the inlets (or outlets), implying a cycle.

Subpartitioning ensures that the second part of the condition for a thread is satisfied.

The partitioning algorithm for a basic block consists of iterated demand and dependence partitioning (with subpartitioning) until the partitions of the basic block do not change. An example of iterated partitioning is given in figure 1.4.

Global Partitioning

In this section, we give a very brief overview of the global analysis presented in [37]. The idea is simple: given a block B calling a block B' we annotate the call site in B using the inlets/outlets of B' and use the new inlets/outlets to partition B. We then propagate the information about the new partitions of B to other blocks calling B. The inlet/outlets of B' are α-renamed so that they do not conflict with the inlet/outlets of B. When a certain dependence is detected between an input and an output, a *squiggly* edge is added to note this dependence.

Information contained in inlets/outlets is characterized mathematically by a *congruence syndrome.*

Definition 1.4 (Congruence Syndrome) *The congruence syndrome of a collection of sets of names $C = S_1, \ldots, S_n$ is a $2^n \times 2^n$ matrix defined as follows:*

$$Syn(C) : Pow(\{1, \ldots, n\}) \times Pow(\{1, \ldots, n\})$$

where $Syn(C)(I_1, I_2) = 1$ iff

$$\bigcup_{i \in I_1} S_i = \bigcup_{i \in I_2} S_i$$

Two inlet sets are "eqvivalent" (in the sense that they produce same partitions if they are propagated) if their congruence syndromes are equal. Note that an α-renaming of a collection of inlets gives a set of inlets having the same congruence syndrome. A partial order on congruence syndromes can be defined as follows:

$$syn_1 \sqsubseteq syn_2 \Longleftrightarrow syn_1(I_1, I_2) \leq syn_2(I_1, I_2) \forall I_1, I_2$$

Consider a block B calls two blocks B_1' and B_2' arising in the case of a conditional, a multiple-def single-use interface. In this case, the algorithm should propagate the *lower bound* of congruence syndromes of the inlets/outlets. It is proved that this can be achieved by taking a union of the inlets and outlets (making sure that the names do not clash).

It is pointed out in [37] that DD partitioning propagates information about non-strict arguments too.

```
f(x,y,z)=
{a =  if x  then {b = y + z  in b}  else { in 2}
 in
   a}
```

In the above example, y and z get the same outlet annotations (as they are used together). In a call-site $f(e_1, e_2, e_3)$, the computation corresponding to e_2 and e_3 can be put in the same thread. This information is not present in the strictness properties of the function (it would just say that f is strict in x and not strict in y and z).

Recursion: Problems

Here, we outline one limitation of the global analysis, that is, its ability to handle recursion. Recursion in a program shows up as a cycle in the call graph. Propagating information around a complete cycle will permanently invalidate all site annotations on the cycle, preventing further progress.

However, we could use fix point iteration (that is, continue propagating around the cycle) until the site annotations have been revalidated. Unfortunately checking for fixed point is difficult. Two wrong ways of checking for fixed point are:

1. Checking to see if the inlets/outlets at the send/receive nodes are equal. However, we generate new labels each time we propagate to ensure safety. Thus, the label sets grow *ad infinitum* and the fix point iteration will not terminate.

2. Checking to see if the partitions of all blocks in the cycle are same. This test is not safe as the information contained in the inlets/outlets could still be changing even though the partitions do not change after an iteration.

The only safe method of checking for fixed point is to check that the congruence syndromes corresponding to the label sets have converged. Though this check is safe, it involves a lot of computation (the sizes of the congruence syndromes are exponential). If we accept a rise in the complexity of the algorithm, we can design a better partitioning algorithm as the rest of the thesis shows.

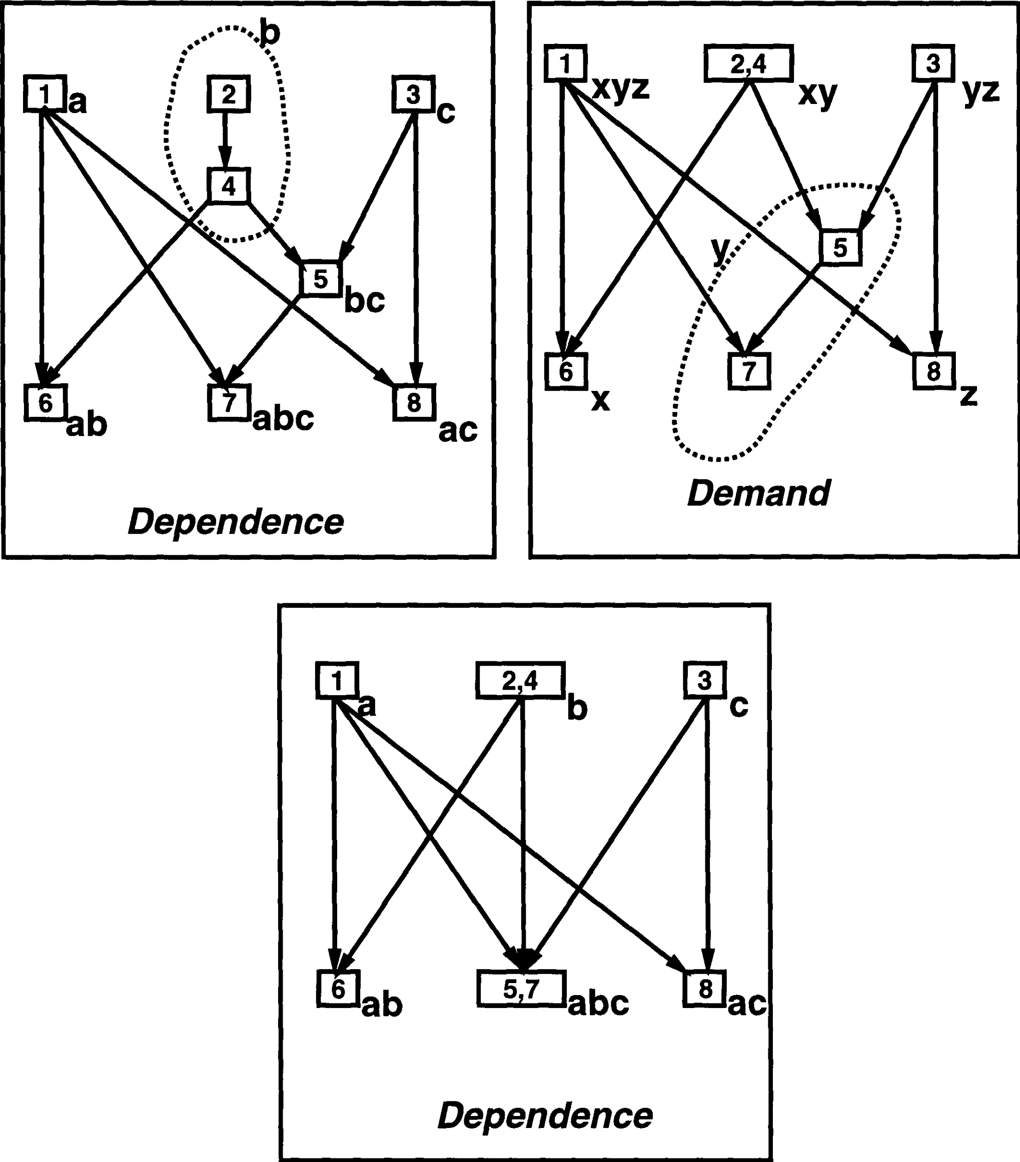


Figure 1.4: Iterated Partitioning

Chapter 2

SP-TAC: Syntax and Semantics

In section 1.3.3 we introduced the *structured dataflow graph* intermediate form into which Id can be compiled. However, we choose to work with another intermediate language (P-TAC [4]) for the following reasons:

1. It is also an intermediate form used in a compiler for Id [46], and techniques for compiling Id to P-TAC are well known.

2. Unlike dataflow graphs, there is a formal operational semantics given for P-TAC. We use this to prove correctness of our algorithms.

3. Like dataflow graphs, P-TAC has minimal "clutter". Thus, we can work with a small, yet powerful language without getting into the details of syntax of Id.

Observe that the DD analysis given in Chapter 1 is essentially an analysis on a first order functional language. It models side-effects as new *outputs* and makes conservative decisions when dealing with data-structures and higher order functions. Thus, to study issues related to the DD algorithm and its problems with recursion we need not directly work with P-TAC (which is a higher-order language with arbitrary data-structures). We start with a subset of it (called Small P-TAC or SP-TAC), which is a first order, functional language with simple data types. SP-TAC does not have structured data-types, but functions can return more than one value. In a later chapter, we will add other features into the language. However, it should be noted that even though we are not dealing with all the features, the techniques that are developed for SP-TAC are valid for P-TAC too; we might have to make conservative approximations when dealing with other features (similar to the approach taken by the DD algorithm).

2.1 Abstract syntax of the language

The abstract syntax of the language is given below:

$c \in Con$	Constants, $c ::= \mathbf{1} \mid \mathbf{2} \mid \mathbf{true} \mid \mathbf{false} \mid \ldots$
$x, y \in V$	Variables
$p \in Pf$	Primitive functions, $p ::= + \mid - \mid \mathbf{and} \mid \ldots$
$f \in Fv$	User defined functions
$se \in SE$	Simple exp, where $se ::= c \mid x$
$e \in Exp$	Expressions, where
	$e ::= se \mid b \mid p(se_1, se_2, \ldots, se_n) \mid$
	$f(se_1, se_2, \ldots, se_n) \mid \mathbf{if}\ se\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$
$b \in Bo$	Blocks, where $b ::= \{\{st\}^*\ \mathbf{in}\ (se_1, se_2, \ldots, se_n)\}$
$st \in St$	Statements, where $st ::= (y_1, y_2, \ldots, y_n) = e$
$pr \in Pr$	Programs, where $P ::= \{f_i(x_1, x_2, \ldots, x_n) = b_i\}$

We make the following assumptions.

1. All primitive operators return a single value as their result.
2. All nested lambda abstractions have been lifted to the top level [23].
3. All functions (user defined and primitive) are not curried. This ensures that the syntax specifies a first order language.
4. All local and bound variables are α-renamed so that they are defined uniquely in the program.

In the example programs given, we take the liberty of using primitive operators like $+, -$ in their infix form, instead of the prefix form specified in the abstract syntax.

2.2 Notation

Given a block b of SP-TAC, the (top level) variables bound in the block ($BV(b)$) is defined as follows:

$$BV(\{\{st_i\}^*\ \mathbf{in}\ (se_1, \ldots, se_m)\}) = \bigcup_i BV(st_i)$$

$$BV((y_1, \ldots, y_n) = e) \quad = \quad \{y_1, \ldots, y_n\}$$

Given an expression e of SP-TAC the *free variables* of e (denoted by $FV(e)$) is defined as follows:

$$\begin{aligned}
FV(c) &= \{\} \\
FV(x) &= \{x\} \\
FV(\{\{st_i\}^* \textbf{ in } (se_1, \ldots, se_m)\}) &= \bigcup_i FV(st_i) \bigcup_{1 \le j \le m} FV(se_j) \setminus \bigcup_i BV(st_i) \\
FV(p(se_1, \ldots, se_n)) &= \bigcup_{1 \le i \le n} FV(se_i) \\
FV(f(se_1, \ldots, se_n)) &= \bigcup_{1 \le i \le n} FV(se_i) \\
FV(\textbf{if } se \textbf{ then } b_1 \textbf{ else } b_2) &= FV(se) \cup FV(b_1) \cup FV(b_2) \\
FV((y_1, \ldots, y_n) = e) &= FV(e) \setminus \{y_1, \ldots, y_n\}
\end{aligned}$$

A substitution (φ) is a mapping from variables to variables. We use $Domain(\varphi)$ to denote the domain of φ. Substitution on an expression is defined as follows:

$$\begin{aligned}
\varphi(c) &= c \\
\varphi(x) &= \varphi(x), x \in Domain(\varphi) \\
&= x, \text{otherwise} \\
\varphi(\{\{st_i\}^* \textbf{ in } (se_1, \ldots, se_m)) &= \{\{\varphi(st_i)\}^* \textbf{ in } (\varphi(se_1), \ldots, \varphi(se_m))\} \\
\varphi(p(se_1, \ldots, se_n)) &= p(\varphi(se_1), \ldots, \varphi(se_n)) \\
\varphi(f(se_1, \ldots, se_n)) &= f(\varphi(se_1), \ldots, \varphi(se_n)) \\
\varphi(\textbf{if } se \textbf{ then } b_1 \textbf{ else } b_2) &= \textbf{if } \varphi(se) \textbf{ then } \varphi(b_1) \textbf{ else } \varphi(b_2) \\
\varphi((y_1, \ldots, y_n) = e) &= (y_1, \ldots, y_n) = \varphi(e)
\end{aligned}$$

A substitution is denoted by $[x_1/y_1, \ldots, x_n/y_n]$, where x_i is substituted for y_i (for $1 \le i \le n$). We sometimes use $[op(x)/x]$ to denote the substitution $op(x)$ for x for all variables x. op is an operator which produces "substitution" variables for each x.

We use $env = [x_1 \mapsto v_1, \ldots, y_1 \mapsto v_n]$ to denote an environment mapping variables $\{x_1, \ldots, x_n\}$ to values $\{v_1, \ldots, v_n\}$. The domain of the environment $(Domain(env))$ is $\{x_1, \ldots, x_n\}$. We will use $[x_i \mapsto v_i]$ to denote an environment mapping x_i to v_i where i ranges over some set determined by the context in which we use the environment.

If env and env' are two environments, we use $env.env'$ to denote the environment env'' such that $env''[\![y]\!] = env[\![y]\!]$ if $y \in Domain(env)$ else $env''[\![y]\!] = env'[\![y]\!]$. This is the way environments are "linked" in a block structured language.

2.3 Operational Semantics

The operational semantics of SP-TAC is given in terms of an Abstract Reduction System [24], which is a structure $\langle A, \longrightarrow_R \rangle$ where A is a set of terms and $\longrightarrow_R$ is a binary relation on A. The operational semantics is directly distilled from the operational semantics of P-TAC given in [4]. A contains all closed SP-TAC terms (terms without free variables), with Program as start symbol, plus all terms that can appear during a program evaluation. The term *ground values* refers to values of SP-TAC, i.e., terms which cannot be reduced any further (integers and booleans). This system is derived from a set of rewrite rules. The operational meaning of user defined functions is given by the user via the program Pr.

2.3.1 Canonical Representation of Terms

Consider the following terms:

1. {x = 8; z = { y = x; u = x + y **in** u} **in** z}
2. {x = 8; z = x + x **in** z}
3. { **in** 8 + 8}

Though these terms are syntactically distinct from each other, they all behave the same operationally. For example, the Id compiler would represent all these terms using the same dataflow graph, i.e., $8 + 8$. Therefore, in our reduction system, terms that have the same graph, up to isomorphism, are equivalent. We give a canonicalization procedure for terms in order to select a representative of terms which are *equivalent. Reductions will be performed only on canonical terms.*

Definition 2.1 (Canonical terms) *The canonical form of a term M is obtained as follows:*

1. *Flatten all blocks according to the following rule:*

$$\{st_1; st_2; \ldots; (x_1, \ldots, x_n) = \{st_{11}; \ldots \textbf{ in } (y_1, \ldots, y_n)\}; \ldots\} \longrightarrow_{blk}$$

$$\{st_1; st_2; \ldots; \varphi(st_{11}); \ldots; (x_1, \ldots, x_n) = (\varphi(y_1), \ldots, \varphi(y_n)); \ldots\}$$

where φ renames the variables in the inner block to new variables (to avoid name clashes).

2. *For each binding of the form $(y_1, \ldots, y_n) = (x_1, \ldots, x_n)$ in M, where x_i's and y_i's are distinct variables, replace each occurrence of y_i in M by x_i for each $1 \leq i \leq n$. Remove the statement $(y_1, \ldots, y_n) = (x_1, \ldots, x_n)$ from M.*

3. *For each binding of the form $(y_1, \ldots, y_n) = (v_1, \ldots, v_n)$ in M, where v_i's are ground values, replace each occurrence of y_i in M by v_i for each $1 \leq i \leq n$. Remove the statement $(y_1, \ldots, y_n) = (v_1, \ldots, v_n)$ from M.*

Definition 2.2 (α-equivalence) *Two closed terms M and N in canonical form are said to be α-equivalent if each can be transformed into the other by a consistent renaming of locations and bound variables.*

Lemma 2.1 *Each SP-TAC term has a unique canonical form up to α-renaming [4].*

2.3.2 Rewrite Rules of SP-TAC

Intuitively, we define the evaluation of a program M as consiting of repeatedly rewriting its subterms until no further rewriting is possible. Some rules have a *precondition* to be satisfied. We apply the rewrite rules only to terms in canonical form. Also, we apply the rewrite rules to the statements in the outermost block only. This prevents terms inside the **then** and **else** blocks of an **if** statement from getting rewritten. A *context* $C[]$ is a term with a hole in it, such that, when a suitable term is plugged in the hole, $C[]$ becomes a proper term [7].

We now present the set of rewrite rules, R_{SP-TAC}, for defining the ARS for SP-TAC. In this section n and $\widehat{n}$ represent a variable and a numeral, respectively. We do not discuss type errors in the rules, we assume that a primitive function is only applied to arguments of appropriate types.

- **δ rules**

$$\begin{aligned}
\widehat{m} + \widehat{n} &\longrightarrow_\delta \widehat{m+n} \\
eq?\ \widehat{m}\ \widehat{m} &\longrightarrow_\delta true \\
eq?\ \widehat{m}\ \widehat{n} &\longrightarrow_\delta false \text{ if } \widehat{m} \neq \widehat{n}
\end{aligned}$$

- **Conditional rules**

$$\textbf{if } true \textbf{ then } b_1 \textbf{ else } b_2 \longrightarrow_{cond} b_1$$
$$\textbf{if } false \textbf{ then } b_1 \textbf{ else } b_2 \longrightarrow_{cond} b_2$$

- **Application rule**

$$\frac{f(x_1,\ldots,x_n) = \{\{st\}^*\textbf{in } (se_1,\ldots,se_m)\} \in Pr}{f(y_1,\ldots,y_n) \longrightarrow_{app} \{x'_1 = y_1;\ldots,x'_n = y_n; \{st'\}^* \textbf{ in } (se'_1,\ldots,se'_m)\}}$$

Note that we have to rename every variable in body of the function to new variables.

Discussion:

1. Non-strictness of the conditional statment arises from the fact that we do not require that all the arguments to the conditional (containing $FV(b_1)$, $FV(b_2)$) should be defined. The rewrite rule holds as soon as the predicate is evaluated.

2. Non-strictness of the function call arises from the fact that we do not insist that the arguments to the function f be ground values. The rewrite rule applies even though the arguments are variables.

2.3.3 Observable results

We now define the notion of an *Answer* of a term. This helps us to reason about equivalence of blocks in SP-TAC and is useful to define correctness of our partitioning transformations given in later chapters.

Definition 2.3 (Answer of a term) *Given an ARS $\langle A, R_{SP-TAC}\rangle$ and $M \in$ Initial-Terms of A, the answer produced by M ($Ans(M)$), is undefined if M does not have a normal form. Otherwise, M reduces to a normal form N, and $Ans(M)$ is*

1. *If N is of the form $\{\textbf{in } v\}$, then $\langle$proper-termination, $v\rangle$.*

2. *If N is of the form $\{st_1; st_2; \ldots \textbf{ in } v\}$ then*

   (a) *If v is either an integer or boolean or* **error**, *then $\langle$improper-termination, $v\rangle$.*

   (b) *else $\langle$improper-termination, Nothing$\rangle$.*

where proper-termination, improper-termination, *and* Nothing *are reserved constants.*

2.4 Denotational Semantics

In addition to operational semantics, we also make use of denotational semantics to derive dependence information. Denotational semantics is a convenient framework to express fix point iteration, which is used to deal with recursive functions. Though the proofs given in the thesis make use of an *equivalence* between the two semantics, we do not attempt a formal proof of this fact. The denotational semantics is based on the standard semantics given in [9].

As functions in our language can have many arguments and return many results, we need a model to capture *multiple values*. We use D^n to denote a domain formed by the n-product $D \times D \times \ldots \times D$ with the usual ordering on the elements. The ordering is:

$$(u_1, \ldots, u_n) \sqsubseteq (v_1, \ldots, v_n) \iff \forall i, 1 \leq i \leq n, u_i \sqsubseteq v_i$$

The infix operator $\downarrow$ is used to select any "component" of a value in D^n.

The semantic domains are:

Int	the standard flat domain of integers.
$Bool$	the standard flat domain of boolean values.
$Val = Int + Bool$	the domain of values.
$Fun = \bigcup_{n\geq 1, m\geq 1}(Val^n \rightarrow Val^m)$	the domain of first order functions.
$D = Val + \{error\}$	the domain of denotable values.
$D_T = \bigcup_{n\geq 1} Val^n + \{error\}$	tuples of values.
$Bve = V \rightarrow D$	the domain of variable environments.
$Fenv = Fv \rightarrow Fun$	the domain of function environments.

The semantic functions are:

$\mathcal{E}_c : Con \rightarrow Val$

$\mathcal{E}_k : Pf \rightarrow Fun$

$\mathcal{E}_s : SE \rightarrow Bve \rightarrow D$

$\mathcal{E}_b : Bo \rightarrow Bve \rightarrow Fenv \rightarrow D_T$

$\mathcal{E} : Exp \rightarrow Bve \rightarrow Fenv \rightarrow D_T$

$\mathcal{E}_p : Prog \rightarrow Fenv$

In the semantic equations, we make use of a case operator (denoted by $\rightarrow$). The meaning of $x \rightarrow y, z$ is that depending whether x is $true$ or $false$ either y or z is selected. The semantic equations are:

$$\begin{aligned}
\mathcal{E}_c[\![n]\!] &= n, \text{integer } n \\
\mathcal{E}_c[\![true]\!] &= true \\
\mathcal{E}_c[\![false]\!] &= false \\
\mathcal{E}_k[\![+]\!] &= \lambda(x,y).(Int?(x)\ and\ Int?(y)) \rightarrow x+y, error \\
\mathcal{E}_s[\![c]\!]\ bve &= \mathcal{E}_c[\![c]\!] \\
\mathcal{E}_s[\![x]\!]\ bve &= bve[\![x]\!] \\
\mathcal{E}[\![se]\!]\ bve\ fenv &= \mathcal{E}_s[\![se]\!]\ bve \\
\mathcal{E}[\![p(se_1, se_2, \ldots, se_n)]\!]\ bve\ fenv &= \mathcal{E}_k[\![p]\!]\ (\mathcal{E}_s[\![se_1]\!]\ bve, \ldots, \mathcal{E}_s[\![se_n]\!]\ bve) \\
\mathcal{E}[\![f(se_1, se_2, \ldots, se_n)]\!]\ bve\ fenv &= fenv[\![f]\!]\ (\mathcal{E}_s[\![se_1]\!]\ bve, \ldots, \mathcal{E}_s[\![se_n]\!]\ bve) \\
\mathcal{E}[\![\textbf{if}\ se\ \textbf{then}\ b_1 \textbf{else}\ b_2]\!]\ bve\ fenv &= (\mathcal{E}_s[\![se]\!]\ bve) \rightarrow (\mathcal{E}_b[\![b_1]\!]\ bve\ fenv), (\mathcal{E}_b[\![b_2]\!]\ bve\ fenv) \\
\mathcal{E}[\![b]\!]\ bve\ fenv &= \mathcal{E}_b[\![b]\!]\ bve\ fenv
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}_b[\![\{\{(y_{i1}, \ldots, y_{im}) = e_i\}^*\ \textbf{in}\ (se_1, se_2, \ldots, se_n)\}]\!]\ bve\ fenv = \\
&\qquad \text{letrec } newbve = [y_{ij} \mapsto (\mathcal{E}[\![e_i]\!]\ (newbve.bve)\ fenv) \downarrow j] \\
&\qquad \text{in } (\mathcal{E}_s[\![se_1]\!]\ newbve, \ldots, \mathcal{E}_s[\![se_n]\!]\ newbve)
\end{aligned}$$

$$\mathcal{E}_p[\![\{f_i(x_1, x_2, \ldots, x_n) = b_i\}^*]\!] = fenv \text{ whererec}$$

$$fenv = [f_i \mapsto (\lambda(y_1, y_2, \ldots, y_n).\mathcal{E}_b[\![b_i]\!]\ [x_i \mapsto y_i]\ fenv)]$$

Chapter 3

Analyzing Recursion using Paths

We have already seen (Chapter 1) that DD partitioning is not able to handle recursion satisfactorily. We have also noted that strictness analysis has no such problems, it is able to deal with recursion using fix point iteration. However, it is not as powerful as one would want; there are functions where better partitions can be obtained using demand dependence.

In this chapter, we use *Paths*, as a way of representing dependence information. *Paths* are very similar to the *strictness sets* defined in [18], and different from the *Paths* in [9], but we continue to use the same terminology. We combine the strong points of both strictness analysis and DD, i.e., ability to handle recursion as well as retain information about non-strict arguments. One problem with DD is that it tries to derive dependence information and do partitioning in a single framework. This gives rise to difficulties when dealing with recursion. We separate the two parts: dependence information is computed and represented using *paths* and this information is used in the partitioning algorithm. Our approach is outlined in figure 3.1.

The rest of the chapter is organized as follows. First, we give a formal definition of path semantics. We then "abstract" the path semantics to compute safe approximations to the paths at compile time. Then, we use paths to generate dependence information (in the form of inlets/outlets) for use in the DD algorithm. Finally, we provide examples on which even the modified DD algorithm does not yield satisfactory partitions.

3.1 Path Semantics

Intuitively, a path of an expression is the set of free variables of that expression required to produce a value for that expression. The path of an expression depends on the run-

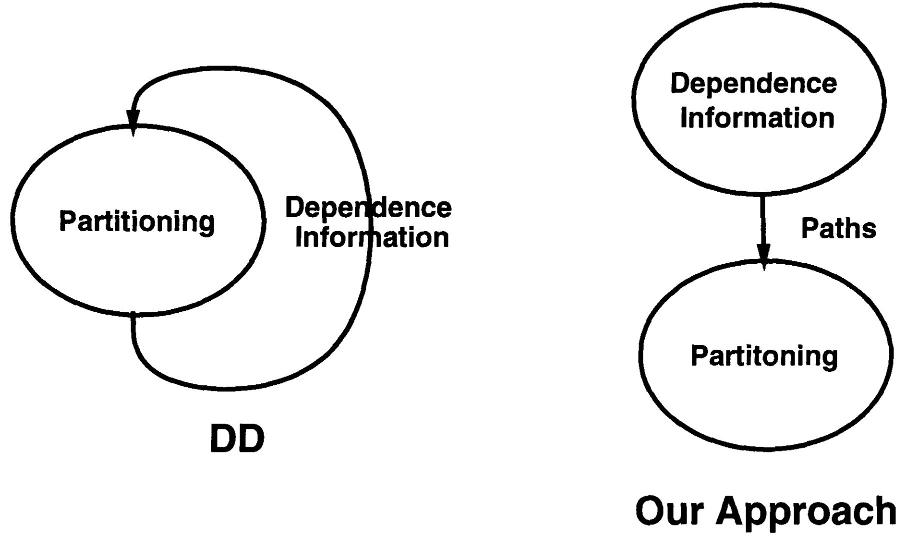


Figure 3.1: Different approaches to Partitioning

time values of variables themselves. For example, depending on the value of the predicate, variables used in one of the arms of a conditional will not be required to produce a value for that conditional. Thus, path semantics needs to use the standard semantics (the run-time values of expressions) to produce paths for a program.

Let *Path* be the flat domain of paths, with $\perp_p$ representing non-termination of the expression. Any two terminating paths are considered incomparable, and non-termination is weaker than any form of termination. *Path* is defined by (see figure 3.2):

$$Path = \{\perp_p\} \bigcup \{[z_1, z_2, \ldots, z_n] | n \geq 0, \forall i, 1 \leq i \leq n, z_i \in V\}$$

A variable x belongs to a path p ($x \in p$) if $p = \perp_p$ or $x \in p$. This reflects the intuition that an expression returning $\perp$ can be made to depend on any variable without loss of safety.

To define paths in an environment, we need to define substitution by paths. Substitution is accomplished by using the modified union operator ":" on paths. Note that ":" is both commutative and associative (like the usual set union operator).

$$\begin{aligned} &\forall p, q \in Path \\ &\quad p : \perp_p = \perp_p \\ &\quad \perp_p : p = \perp_p \\ &\quad p : q = p \cup q \text{ if } p \neq \perp_p \text{ and } q \neq \perp_p. \end{aligned}$$

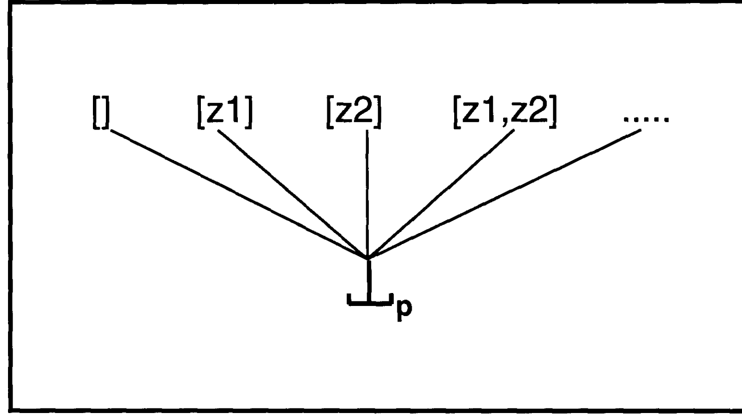


Figure 3.2: Structure of the Path Domain

We now give a "non-standard" semantics that gives information about expressions: not only the values of the expressions but also the paths corresponding to expressions. For clarity, we have omitted the "checking" for errors by primitive operators in this semantics. The semantic domains are:

Val, D, the domains of values and denotable values (from standard semantics)

$Path$, the flat domain of paths

$Pfun = \bigcup_{n \geq 1, m \geq 1}((D \times Path)^n \rightarrow (D \times Path)^m)$, the path functions

$DP_T = \bigcup_{n \geq 1}(D \times Path)^n$, multiple value-path pairs

$Penv = Fv \rightarrow Pfun$, the function environment

$Pbve = V \rightarrow (D \times Path)$, variable environment

The semantic functions are:

$\mathcal{E}_c : Con \rightarrow Val$, semantic function for constants

$\mathcal{P} : Exp \rightarrow Pbve \rightarrow Penv \rightarrow DP_T$

$\mathcal{P}_k : Pf \rightarrow Pfun$

$\mathcal{P}_p : Prog \rightarrow Penv$

$\mathcal{P}_b : Bo \rightarrow Pbve \rightarrow Penv \rightarrow DP_T$

$\mathcal{P}_s : SE \rightarrow Pbve \rightarrow (D \times Path)$

$$\begin{aligned} \mathcal{P}_k[\![+]\!] &= \lambda((x_e, x_p), (y_e, y_p)).\,(x_e + y_e), (x_p : y_p) \\ \mathcal{P}_s[\![c]\!]\, pbve &= (\mathcal{E}_c[\![c]\!], []) \end{aligned}$$

$$\mathcal{P}_s[\![x]\!]\, pbve \;=\; pbve[\![x]\!]$$

$$\begin{aligned}
\mathcal{P}[\![se]\!]\, pbve\, penv &= \mathcal{P}_s[\![se]\!]\, pbve \\
\mathcal{P}[\![p(se_1,\ldots,se_n)]\!]\, pbve\, penv &= \mathcal{P}_k[\![p]\!](\mathcal{P}_s[\![se_1]\!]\, pbve,\ldots,\mathcal{P}_s[\![se_n]\!]\, pbve) \\
\mathcal{P}[\![f(se_1,\ldots,se_n)]\!]\, pbve\, penv &= penv[\![f]\!](\mathcal{P}_s[\![se_1]\!]\, pbve,\ldots,\mathcal{P}_s[\![se_n]\!]\, pbve) \\
\mathcal{P}[\![b]\!]\, pbve\, penv &= \mathcal{P}_b[\![b]\!]\, pbve\, penv
\end{aligned}$$

$$\begin{aligned}
&\mathcal{P}[\![\textbf{if } se \textbf{ then } b_1 \textbf{ else } b_2]\!]\, pbve\, penv = \\
&\qquad \text{let } (x_e, x_p) = \mathcal{P}_s[\![se]\!]\, pbve \\
&\qquad\quad ((y_{e1}, y_{p1}),\ldots,(y_{en}, y_{pn})) = \mathcal{P}_b[\![b_1]\!]\, pbve\, penv \\
&\qquad\quad ((z_{e1}, z_{p1}),\ldots,(z_{en}, z_{pn})) = \mathcal{P}_b[\![b_2]\!]\, pbve\, penv \\
&\qquad \text{in } x_e \rightarrow ((y_{e1}, x_p : y_{p1}),\ldots,(y_{en}, x_p : y_{pn})), ((z_{e1}, x_p : z_{p1}),\ldots,(z_{en}, x_p : z_{pn}))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{P}_b[\![\{\{(y_{i1},\ldots,y_{im}) = e_i\}^* \textbf{ in } \{se_1,\ldots,se_n\}\}]\!]\, pbve\, penv = \\
&\qquad \text{let } newpbve = [y_{ij} \mapsto (\mathcal{P}[\![e_i]\!]\,(newpbve.pbve)\, penv) \downarrow j] \\
&\qquad \text{in } (P_s[\![se_1]\!]\, newpbve,\ldots,P_s[\![se_n]\!]\, newpbve)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{P}_p[\![\{f_i(x_1,\ldots,x_n) = b_i\}^*]\!] = penv, \text{ whererec} \\
&penv = [f_i \mapsto (\lambda((e_1,p_1),\ldots,(e_n,p_n)).\mathcal{P}_b[\![b_i]\!]\,[x_i \mapsto (e_i,p_i)]\, penv)]
\end{aligned}$$

Discussion:

1. The semantic function $\mathcal{P}_s$ (for constants and variables) is straightforward. In case of a constant, it returns the empty path, reflecting the fact that a constant does not have to depend on any variable for its value. For variables, it returns the value present in the environment.

2. For primitive functions, the paths returned reflect the *strict* behavior of primitive functions. For the "+" function, a union of the variables of the first operand and the second operand is returned modeling the fact that "+" requires both operands to produce a value.

3. The semantic function for **if** returns one of two paths (depending upon the value of the predicate), one corresponding to the variables needed when the **then** branch is taken and the other corresponding to the variables needed when the **else** branch is taken.

4. The semantic function $\mathcal{P}$ for expressions recurses appropriately when the expression is either a primitive function or a user defined function.

5. The semantic function for blocks uses recursion in the environment to model recursion in the block.

6. The semantic definition for the program uses recursion in the function environment to model recursion among functions. The fix point analogue of this definition would start with a function environment mapping all functions to the undefined function. Each iteration would involve evaluating the function bodies using this environment, and updating the function environment using values obtained. We continue the process until fix point is reached.

7. It is easy to show that $\mathcal{P}$ satisfies the usual monotonicity property of semantic functions. That is,

$$(pbve' \sqsubseteq pbve) \wedge (penv' \sqsubseteq penv) \Longrightarrow (\mathcal{P}[\![e]\!]\, pbve'\, penv' \sqsubseteq \mathcal{P}[\![e]\!]\, pbve\, penv)$$

   This follows from the monotonicity of the : operator.

We now prove a theorem that our path semantics is *consistent* with the standard semantics. Basically, we are saying that if the path semantics gives a path for the expression,

1. All the variables present in the path are *needed* to produce a value for the expression.

2. No other variable is needed to produce the value for the expression.

Before we proceed to state the theorem, we need the following definition of a *projection* of an environment with respect to a path.

Definition 3.1 (Projection of environments) *Given an environment bve mapping variables to values, and a path p, the* projection $(Proj(bve, p, pbve))$ *of bve with respect to p and a path environment pbve is an environment bve′ such that:*
$bve'[\![x]\!] = bve[\![x]\!]$, *if* $(v, p_x) = pbve[\![x]\!]$ *and* $p_x \subseteq p$
$bve'[\![x]\!] = \bot$, *otherwise.*

This operation converts the environment *bve* to another environment *bve′* such that $bve' \sqsubseteq bve$, retaining values of variables whose paths are a subset of p.

Theorem 3.1 (Consistency of Path semantics) *Let pr be a program with* $fenv = \mathcal{E}_p[\![pr]\!]$ *and* $penv = \mathcal{P}_p[\![pr]\!]$. *Let e be an expression and bve be an environment mapping variables to values. Let pbve be an environment mapping variables in Domain(bve) to value-path pairs such that*

1. $pbve[\![y]\!] = (bve[\![y]\!], p_y)$.

2. *If* $bve[\![y]\!] \neq \perp$, *then* $p_y \neq \perp_p$ *and* $\forall x \in p_y, bve[\![x]\!] \neq \perp$.

Then, $\mathcal{E}[\![e]\!] bve\, fenv = (v_1, \ldots, v_n)$ *implies* $\mathcal{P}[\![e]\!] pbve penv = ((v_1, p_1), \ldots, (v_n, p_n))$. *Further, for each* $i = 1 \ldots n$,

1. *(A) if* $v_i \neq \perp$ *then* $p_i \neq \perp_p$ *and* $\forall x \in p_i, bve[\![x]\!] \neq \perp$.

2. *(B)* $(\mathcal{E}[\![e]\!]\, Proj(bve, p_i, pbve)\, fenv) \downarrow i = v_i$.

Proof: Note that condition (B) is always satisfied if $v_i = \perp$. This follows from the fact that $\mathcal{E}$ is monotonic, and $Proj(bve, p_i, pbve) \sqsubseteq bve$. Thus, we prove (B) only for the case when $v_i \neq \perp$. The proof is by straightforward structural and fixpoint induction. We enumerate the cases of e:

1. $e = se$. There are two cases:

   (a) $se = c$: $\mathcal{E}_s[\![e]\!]\, bve = \mathcal{E}_c[\![c]\!]$. $\mathcal{P}_s[\![e]\!]\, pbve = (\mathcal{E}[\![c]\!], [])$. (A) is obviously true. (B) follows from

   $$\mathcal{E}_s[\![c]\!]\, Proj(bve, [], pbve) = \mathcal{E}_c[\![c]\!]$$

   (b) $se = x$: $\mathcal{E}_s[\![x]\!]\, bve = bve[\![x]\!]$. $\mathcal{P}_s[\![x]\!]\, pbve = pbve[\![x]\!] = (bve[\![x]\!], p_x)$ by definition of $pbve$. (A) follows directly from the condition on $pbve$. To prove (B), note that $\mathcal{E}_s[\![x]\!]\, Proj(bve, p_x, pbve) = Proj(bve, p_x, pbve)[\![x]\!] = bve[\![x]\!]$ as $pbve[\![x]\!] = (bve[\![x]\!], p_x)$ and $p_x \subseteq p_x$.

2. $e = p(se_1, \ldots, se_n)$. We prove this for the "+" primitive operator.

$\mathcal{E}[\![+(se_1, se_2)]\!]\, bve\, fenv =$

$$\begin{aligned} &\text{let } se_{v1} = \mathcal{E}_s[\![se_1]\!]\, bve \\ &\quad\;\; se_{v2} = \mathcal{E}_s[\![se_2]\!]\, bve \\ &\text{in } se_{v1} + se_{v2}. \end{aligned}$$

$\mathcal{P}[\![+(se_1, se_2)]\!]\, pbve\; penv =$

$\quad$ let $(se_{e1}, se_{p1}) = \mathcal{P}_s[\![se_1]\!]\, pbve$

$\quad\quad (se_{e2}, se_{p2}) = \mathcal{P}_s[\![se_2]\!]\, pbve$

$\quad$ in $(se_{e1} + se_{e2}, se_{p1} : se_{p2})$.

By structural induction, $se_{v1} = se_{e1}$, $se_{v2} = se_{e2}$. Thus, $se_{v1} + se_{v2} = se_{e1} + se_{e2}$. Let $se_{v1} + se_{v2} \neq \bot$. Then, $se_{v1} \neq \bot$. Thus, $se_{p1} \neq \bot_p$ and $\forall x \in se_{p1}, bve(x) \neq \bot$ (by structural induction). A similar property is true for variables $x \in se_{p2}$. Thus, (A) is true for $se_{p1} : se_{p2}$.

To prove (B) define $bve' = Proj(bve, se_{p1} : se_{p2}, pbve)$, $bve'_1 = Proj(bve, se_{p1}, pbve)$ and $bve'_2 = Proj(bve, se_{p2}, pbve)$. As $se_{p1} \subseteq (se_{p1} : se_{p2})$, the definition of $Proj$ implies that $bve'_1 \sqsubseteq bve'$. Similarly, $bve'_2 \sqsubseteq bve'$. By induction $\mathcal{E}_s[\![se_1]\!]\, bve'_1 = se_{v1}$ and $\mathcal{E}_s[\![se_2]\!]\, bve'_2 = se_{v2}$. By monotonicity of $\mathcal{E}$ and the fact that both se_{v1} and se_{v2} are not $\bot$, we have $\mathcal{E}_s[\![se_1]\!]\, bve' = \mathcal{E}_s[\![se_1]\!]\, bve'_1 = se_{v1}$. Similarly $\mathcal{E}_s[\![se_2]\!]\, bve' = se_{v2}$. Thus, $\mathcal{E}[\![+(se_1, se_2)]\!]\, bve' = se_{v1} + se_{v2}$, and (B) is satisfied.

3. $e =$ **if** x **then** b_1 **else** b_2. Let

$$\begin{aligned}
x_v &= \mathcal{E}_s[\![x]\!]\, bve \\
(y_{v1}, \ldots, y_{vn}) &= \mathcal{E}_b[\![b_1]\!]\, bve\; fenv \\
(z_{v1}, \ldots, z_{vn}) &= \mathcal{E}_b[\![b_2]\!]\, bve\; fenv \\
(x_e, x_p) &= \mathcal{P}_s[\![x]\!]\, pbve \\
((y_{e1}, y_{p1}), \ldots, (y_{en}, y_{pn})) &= \mathcal{P}_b[\![b_1]\!]\, pbve\; penv \\
((z_{e1}, z_{p1}), \ldots, (z_{en}, z_{pn})) &= \mathcal{P}_b[\![b_2]\!]\, pbve\; penv
\end{aligned}$$

By structural induction, $x_v = x_e$, $y_{vi} = y_{ei}$, $z_{vi} = z_{ei}$ for all $i \in 1 \ldots n$. If $x_v = \bot$ both the standard and path semantics return $\bot$ and the statements (A) and (B) are satisfied.

Assume x_v is $true$. Then, $\mathcal{E}[\![e]\!]\, bve\; fenv = (y_{v1}, \ldots, y_{vn})$, and $\mathcal{P}[\![e]\!]\, pbve\; penv$ $= ((y_{e1}, y_{p1}), \ldots, (y_{en}, y_{pn})) = ((y_{v1}, y_{p1}), \ldots, (y_{vn}, y_{pn}))$.

The path semantics gives $x_p : y_{pi}$ as the path of the ith component. Using this, we can prove that the (A) and (B) statements are satisfied. The proof is very similar to the one given for the "+" operator.

The proof for the case when x_v is $false$ is similar.

4. $e = \{\{(y_{i1}, \ldots, y_{im}) = e_i\}^* \textbf{ in } (se_1, \ldots, se_n)\}$. We now have to use fix point induction. The environments *newbve* and *newpbve* are limits of the chains $newbve^0$, $newbve^1, \ldots$ and $newpbve^0$, $newpbve^1, \ldots$ where

$$\begin{aligned} newbve^0 &= [y_{ij} \mapsto \bot] \\ newpbve^0 &= [y_{ij} \mapsto (\bot, \bot_p)] \\ newbve^{k+1} &= [y_{ij} \mapsto (\mathcal{E}[\![e_i]\!]\,(newbve^k.bve)\,fenv) \downarrow j] \\ newpbve^{k+1} &= [y_{ij} \mapsto (\mathcal{P}[\![e_i]\!]\,(newpbve^k.pbve)\,penv) \downarrow j] \end{aligned}$$

We show by induction that the following properties are valid for $newbve^k$, $newpbve^k$ for $k \geq 0$.

(a) $newpbve^k.pbve[\![y]\!] = (newbve^k.bve[\![y]\!], p_y)$.

(b) If $newbve^k.bve[\![y]\!] \neq \bot$, then $p_y \neq \bot_p$ and $\forall x \in p_y, newbve^k.bve[\![x]\!] \neq \bot$.

By conditions on *pbve* and *bve*, the two properties are valid when $k = 0$.

Assume that the properties are true for k. We prove the properties for $k+1$. By structural induction, we have:

$$(\mathcal{E}[\![e_i]\!]\ newbve^k.bve\ fenv) \downarrow j = v_j \Longrightarrow (\mathcal{P}[\![e_i]\!]\ newpbve^k.pbve\ penv) \downarrow j = (v_j, p_j)$$

By definition, $newbve^{k+1}[\![y_{ij}]\!] = v_j$ and $newpbve^{k+1}[\![y_{ij}]\!] = (v_j, p_j)$ and the first property is satisfied.

We also have, if $newbve^{k+1}[\![y_{ij}]\!] = v_j \neq \bot$, then $p_j \neq \bot_p$ and $\forall x \in p_j$, $newbve^k.bve[\![x]\!] \neq \bot$. As the *newbve* sequence is monotonic ($newbve^k \sqsubseteq newbve^{k+1}$), $newbve^k.bve[\![x]\!] \neq \bot$ implies $newbve^{k+1}.bve[\![x]\!] \neq \bot$ and the second property is also satisfied.

By fixpoint induction, the properties are true for *newbve* and *newpbve*, the limits of the chains. By structural induction the theorem is satisfied for $(se_1, \ldots, se_n)$, the outputs of the block.

5. $e = f_l(se_1, \ldots, se_n)$. The proof uses fixed point induction. The function environments *fenv* and *penv* are limits of the chains $fenv^0, fenv^1, \ldots$ and $penv^0, penv^1, \ldots$ where

$$\begin{aligned} fenv^0 &= \lambda(y_1, \ldots, y_n).(\bot, \ldots, \bot) \\ penv^0 &= \lambda((e_1, p_1), \ldots, (e_n, p_n)).((\bot, \bot_p), \ldots, (\bot, \bot_p)) \end{aligned}$$

$$
\begin{aligned}
fenv^{k+1} &= [f_l \mapsto (\lambda(y_1,\ldots,y_n).\mathcal{E}_b[\![b_l]\!]\,[x_i \mapsto y_i]\,fenv^k)] \\
penv^{k+1} &= [f_l \mapsto (\lambda((e_1,p_1),\ldots,(e_n,p_n)).\mathcal{P}_b[\![b_l]\!]\,[x_i \mapsto (e_i,p_i)]\,penv^k)]
\end{aligned}
$$

Let $v_i = \mathcal{E}_s[\![se_i]\!]\,bve$ and $(v_i', p_i) = \mathcal{E}_s[\![se_i]\!]\,pbve$. By structural induction,

(a) $v_i = v_i'$.

(b) If $v_i \neq \bot$ then $p_i \neq \bot_p$ and $bve[\![y]\!] \neq \bot$ for $y \in p_i$.

(c) $\mathcal{E}_s[\![se_i]\!]\,Proj(bve, p_i, pbve) = v_i$.

Consider the two expressions

$$
\begin{aligned}
\mathcal{E}[\![f_l(x_1,\ldots,x_n)]\!]\,[x_i \mapsto v_i]\,fenv^k &= (u_1,\ldots,u_m) \\
\mathcal{P}[\![f_l(se_1,\ldots,se_n)]\!]\,[x_i \mapsto (v_i,[x_i])]\,penv^k &= ((u_1',q_1),\ldots,(u_m',q_m))
\end{aligned}
$$

These are identical to the original expressions except that wherever the variable x_i (used in the path environment) occurs in a final path, it should be replace by variables in p_i. Call the new environments bve' and $pbve'$. Note that they to satisfy the conditions of the theorem. We prove the following properties for each k:

(a) $\forall j \in \{1\ldots m\}, u_j = u_j'$.

(b) $u_j \neq \bot$ implies $q_j \neq \bot_p$ and $\forall x \in q_j$, $bve'[\![x]\!] \neq \bot$.

(c) $(\mathcal{E}[\![f_l(x_1,\ldots,x_n)]\!]\,Proj(bve', q_j, pbve')\,fenv^k) \downarrow j = u_j$.

For $k = 0$, $u_j = u_j' = \bot$, and $q_j = \bot_p$, and the properties are obviously satisfied. Assume the properties are true for k. We prove the properties for $k+1$. The two expressions can be written as:

$$
\begin{aligned}
\mathcal{E}_b[\![b_l]\!]\,[x_i \mapsto v_i]\,fenv^k &= (u_1,\ldots,u_m) \\
\mathcal{P}_b[\![b_l]\!]\,[x_i \mapsto (v_i,x_i)]\,penv^k &= ((u_1',q_1),\ldots,(u_m',q_m))
\end{aligned}
$$

By structural and fixpoint induction these properties hold, as the environments satisfy the conditions of the theorem.

Using these properties, it is simple to show that the properties are valid with respect to the environments bve and $pbve$.

□

Using the theorem, we can prove the following corollary. Informally, the corollary states that if the value of an expression in an environment (bve) is not $\bot$, then it is $\bot$ in a "lesser" environment (bve') if and only if some variable needed to produce a value is $\bot$ in bve'.

Corollary 3.1.1 *In theorem 3.1, let* $pbve[\![x]\!] = (bve[\![x]\!], [x])$. *Let* $(\mathcal{E}[\![e]\!] bve\, fenv) \downarrow j = v_j \neq \perp$, $(\mathcal{P}[\![e]\!]\, pbve\; penv) \downarrow j = (v_j, p_j)$. *Let* bve' *be an environment such that* $bve' \sqsubseteq bve$. *Let* $(\mathcal{E}[\![e]\!]\, bve'\, fenv) \downarrow j = u_j$. *Then,*

$$u_j = \perp \Longleftrightarrow \exists x \in p_j, bve'[\![x]\!] = \perp.$$

Proof: Observe that *pbve* and *bve* satisfy conditions of theorem 3.1. Thus, $p_j \neq \perp_p$, and $\forall x \in p_j, bve[\![x]\!] \neq \perp$. Also, $\mathcal{E}[\![e]\!]\; Proj(bve, p_j, pbve)\; fenv = v_j$. Define $pbve'[\![x]\!] = (bve'[\![x]\!], [x])$. There are two cases:

1. $(\Rightarrow) u_j = \perp$. The proof is by contradiction. Assume there is no such x. Then, it follows that $Proj(bve, p_j, pbve) \sqsubseteq bve'$ (by definition of $Proj$). Thus, by monotonicity $v_j = \mathcal{E}[\![e]\!]\; Proj(bve, p_j, pbve)\; fenv$
   $\sqsubseteq \mathcal{E}[\![e]\!]\; bve'\; fenv = u_j$.
   Thus, $v_j \sqsubseteq u_j$, a contradiction.
2. $(\Leftarrow) u_j \neq \perp$. As $bve' \sqsubseteq bve$, we have $u_j = v_j$. Consider $(\mathcal{P}[\![e]\!]\, pbve'\; penv) \downarrow j = (u_j, p_j')$. By theorem 3.1, we have $p_j' \neq \perp_p$. As $\mathcal{P}$ is monotonic, and $pbve' \sqsubseteq pbve$, we have $p_j' \sqsubseteq p_j$. Thus, $p_j' = p_j$. By theorem 3.1, $\forall x \in p_j', bve'[\![x]\!] \neq \perp$, thus $\forall x \in p_j, bve'[\![x]\!] \neq \perp$.

□

3.2 Abstracting Paths

In the previous section, we used the standard semantics (i.e., the "run-time" values of expressions) to give paths for a program. As we cannot do this at compile time, we make a conservative approximation by producing a *set* of paths for an expression. The intuition behind this is that at run time one of the paths will be valid for the expression, but we do not know which.

As *Path* is a domain (i.e. a set with structure), we use a *Powerdomain* construction (instead of the usual *Powerset* construction) to model a set of paths. Unlike the powerset construction there is no "right" powerdomain construction; one has to choose the powerdomain suited for his/her application. See [39] for an introduction to powerdomains. We use the Egli-Milner (EM) Powerdomain construction to model sets of paths, as it is capable of reasoning with sets containing $\perp_p$.

For a flat pointed complete partial order D, the discrete EM powerdomain of D, written $\mathbf{P_{EM}}(D)$, consists of nonempty subsets of D which are either finite, or contain bottom, partially ordered as follows: $\forall A, B \in \mathbf{P_{EM}}(D), A \sqsubseteq_{EM} B$ iff

1. For every $a \in A, \exists b \in B$ such that $a \sqsubseteq_D b$.
2. For every $b \in B, \exists a \in A$ such that $a \sqsubseteq_D b$.

The bottom element of $\mathbf{P_{EM}}(Path)$ is the set $\{\perp_p\}$.

We also need a powerdomain construction for multiple paths, where a set of multiple paths of the form $(p_1, p_2, \ldots, p_n)$ is formed by taking the cross-product of n elements of $\mathbf{P_{EM}}(Path)$. We will use the symbol $\mathbf{P_T}$ to denote this powerdomain of multiple paths. Using the operator $\downarrow$ which produces an element of domain D given an element of the domain D^n, we can define an "overloaded" $\downarrow$ which operates on a set of elements of D^n and returns a set of elements of D, each of which is the ith component of each element of D^n (i is the second operand to $\downarrow$).

We use the following semantic domains:

$Path$,		the domain of paths
PTS	$=$	$\bigcup_{n=1}^{\infty}(\mathbf{P_T}(Path^n))$
		Domain of sets of multiple paths
$\mathbf{P_{EM}}(Path)$,		the powerdomain of $Path$
$Pfun$	$=$	$\bigcup_{n=1}^{\infty} \bigcup_{m=1}^{\infty}(\mathbf{P_T}(Path^n) \rightarrow \mathbf{P_T}(Path^m))$,
		the abstract function space mapping sets of paths to sets of paths
$Aenv$	$=$	$Fv \rightarrow Pfun$,
		the function environment
$Abve$	$=$	$Bv \rightarrow (\mathbf{P_{EM}}(Path))$, the bound variable environment

The semantic functions are:

$\mathcal{A} : Exp \rightarrow Abve \rightarrow Aenv \rightarrow PTS$

$\mathcal{A}_s : SE \rightarrow Abve \rightarrow \mathbf{P_{EM}}(Path)$

$\mathcal{A}_k : Pf \rightarrow Pfun$

$\mathcal{A}_b : Bo \rightarrow Abve \rightarrow Aenv \rightarrow PTS$

$\mathcal{A}_p : Pr \rightarrow Aenv$

The semantic equations are given below.

$$\begin{aligned}
\mathcal{A}_k[\![+]\!] &= \lambda s.\{x : y | (x, y) \in s\} \\
\mathcal{A}_s[\![c]\!]\, abve &= \{[]\} \\
\mathcal{A}_s[\![x]\!]\, abve &= abve[\![x]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{A}[\![se]\!]\ abve\, aenv &= \mathcal{A}_s[\![se]\!]\, abve \\
\mathcal{A}[\![b]\!]\, abve\, aenv &= \mathcal{A}_b[\![b]\!]\, abve\, aenv \\
\mathcal{A}[\![p(se_1, se_2, \ldots, se_n)]\!]\, abve\, aenv &= \mathcal{A}_k[\![p]\!]\, (\mathcal{A}_s[\![se_1]\!]\, abve \times \cdots \times \mathcal{A}_s[\![se_n]\!]\, abve) \\
\mathcal{A}[\![f(se_1, se_2, \ldots, se_n)]\!]\, abve\, aenv &= aenv[\![f]\!]\, (\mathcal{A}_s[\![se_1]\!]\, abve \times \cdots \times \mathcal{A}_s[\![se_n]\!]\, abve)
\end{aligned}$$

$\mathcal{A}[\![\textbf{if}\ se\ \textbf{then}\ b_1\ \textbf{else}\ b_2]\!]\ abve\ aenv =$

let $x \in \mathcal{A}_s[\![se]\!]\ abve$

$(y_1, \ldots, y_n) \in \mathcal{A}_b[\![b_1]\!]\ abve\ aenv$

$(z_1, \ldots, z_n) \in \mathcal{A}_b[\![b_2]\!]\ abve\ aenv$

in $\{(x : y_1, \ldots, x : y_n), (x : z_1, \ldots, x : z_n)\}$.

$\mathcal{A}_b[\![\{\{(y_{i1}, \ldots, y_{im}) = e_i\}^*\ \textbf{in}\ (se_1, se_2, \ldots, se_n)\}]\!]\ abve\ aenv =$

letrec $newabve = [y_{ij} \mapsto (\mathcal{A}[\![e_i]\!]\ (newabve.abve)\ aenv) \downarrow j]$

in $\bigcup_{newabve'} (\mathcal{A}_s[\![se_1]\!]\ newabve' \times \cdots \times \mathcal{A}_s[\![se_n]\!]\ newabve')$ where

$newabve' = [y_{ij} \mapsto \{p_{ij}\}]$, such that $p_{ij} \in newabve[\![y_{ij}]\!]$ and

$p_{ij} \in (\mathcal{A}[\![e_i]\!]\ newabve'\ aenv) \downarrow j$

$\mathcal{A}_p[\![\{f_i(x_1, x_2, \ldots, x_n) = b_i\}]\!] = aenv$ whererec

$aenv = [f_i \mapsto (\lambda s. \bigcup\{\mathcal{A}_b[\![b_i]\!]\ [x_i \mapsto \{y_i\}]aenv | (y_1, \ldots, y_n) \in s\})]$

Discussion: We discuss only those semantic equations which are different from the ones given in the path semantics.

1. The semantic function for **if** returns two sets, one corresponding to the variables needed when the **then** branch is taken and the other corresponding to when the **else** branch is taken.

2. The semantic function for blocks is a little more complicated. First, it uses a recursively defined environment construction (i.e., *newabve*) to mimic the behavior of the

recursively defined block (as in the path semantics). However, as we represent environments as functions between variables to values, we lose correlations between the paths of some variables (in particular, we might lose information when dealing with functions returning more than one value). The *newabve'* construction eliminates such "noise" by additional tests. Consider the following example.

```
{(a,b) = f(...)
 in
   (a,b)}
```

Let the paths for $f(\ldots)$ be $\{(p_1, q_1), (p_2, q_2)\}$. Then, the environment will contain the mappings $[(a \mapsto \{p_1, p_2\}), (b \mapsto \{q_1, q_2\})]$. If we use the environment directly to return the paths of the entire block we would form all the four combinations $\{(p_1, q_1), (p_1, q_2), (p_2, q_1), (p_2, q_2)\}$, of which only two are "valid". We can eliminate the extra paths (like (p_1, q_2)) by making an additional check using the paths for f.

Theorem 3.2 (Effectiveness) *$\mathcal{A}_p[\![pr]\!]$ is computable for any finite program pr.*

Proof: The theorem follows from finite domains and monotonicity of union ($\cup$) and product ($\times$) operators. Proved for a similar language in [9]. □

Complexity: The number of possible paths through a n argument function is $2^n + 1$. This follows from the fact that a path of a function $f(x_1, x_2, \ldots, x_n)$ can either be a subset of $\{x_1, x_2, \ldots, x_n\}$ or $\perp_p$. Thus, path analysis can be done in $O(2^n)$ time. As path analysis subsumes strictness analysis, this is the best we can do. Strictness analysis is Deterministic Exponential Time Complete [18].

Theorem 3.3 (Safety) *Let penv and aenv be computed for a program pr. For any expression e, path environment pbve (mapping $FV(e)$ to value-path pairs) construct $abve[\![x]\!] = \{p_x\}$ where $pbve[\![x]\!] = (v_x, p_x)$. If $((v_1, p_1), \ldots, (v_n, p_n)) = \mathcal{P}[\![e]\!]\, pbve\, penv$, then*

$$(p_1, \ldots, p_n) \in \mathcal{A}[\![e]\!]\, abve\, aenv$$

Proof: The proof is similar to the proof of consistency of path semantics. It uses structural and fixed point induction. As the proof is very similar to the one in [9], we omit the proof here. □

Our partitioning algorithm given later makes use of the paths of an expression; we use paths to characterize dependence behavior of the expression. To obtain paths of an expression with respect to a program, we take the meaning of the expression in a variable environment where every free variable of the expression is bound to itself. This gives information about which of the free variables are needed to produce a value for that expression.

Definition 3.2 (Paths of an expression) *For an expression e, its path (denoted by P(e)) with respect to a program pr is* $\mathcal{A}[\![e]\!]\,[x_i \mapsto \{[x_i]\}]\,(\mathcal{A}_p[\![pr]\!])$, *where* $x_i \in FV(e)$.
We overload the definition of $\in$ *and extend it to work on a set of paths. A variable* $x \in P$ *if* $\exists p \in P$ *such that* $x \in p$.

Example 3.1:
```
f(x,y)={a = x + y
         in
           a}
```

$P(f(x,y)) = \{[x,y]\}$

Example 3.2:
```
f(x,y,z)={a= if x  then { in y}  else { in z}
 in
   a}
```

$P(f(x,y,z)) = \{[x,y],[x,z]\}$

The paths shown for the examples above follow directly from the definitions for + and **if** operators. We are able to represent information more concisely than naive strictness representation [26], which would have represented example 3.2 by a table of size $8(=2^3)$. The frontiers representation [11] does better. However, paths carry more information than contained in traditional strictness analysis. Specifically, they contain information about *correlations* between non-strict input variables, which is not present in strictness analysis.

Example 3.3:
```
fact(n) =
{p = eq? n 0;
 r =  if p  then
                 { in 1}
             else
                 {t1 = n - 1;
                  t2 = fact(t1);
                  t3 = t2 * n
                  in
                    t3}
```

in
`r}`

$P(fact(n)) = \{[n], \perp_p\}$

For the `fact` example, we show the method of computing paths for the function by fix point iteration.

Var	$Paths_0$	$Paths_1$
`p`	$[n]$	$[n]$
`1`	$[]$	$[]$
`t1`	$[n]$	$[n]$
`t2`	$\perp_p$	$[n], \perp_p$
`t3`	$\perp_p$	$[n], \perp_p$
`r`	$[n], \perp_p$	$[n], \perp_p$

For example 1.2, the paths are $P(n) = \{[n], \perp_p\}$. This makes it clear that only *free* variables of the expression are used to compute paths for it. The complexity of the function body (with its cyclic dependencies) is not reflected in the path. Paths represent only the *external* behavior of a function.

3.3 Integrating Paths and DD

We have seen that the inter block propagation part of the DD algorithm makes two changes to the "calling" block:

1. A squiggly edge is introduced between a send and a receive if we can detect that the output (receive) is strict on the input (send).

2. It annotates sends and receives using inlet/outlet annotations present in the "called" block.

Using paths, we are able to derive this information directly:

1. We can detect that an output is strict in an input if the input is present in all the paths of the output.

2. We label (outlets of) each send with names of paths in which the corresponding input is present.

Our method of propagating inlets to receives is weaker than the method used by the DD algorithm. We are unable to represent correlations between two outputs, which show up as identical inlet sets in the DD algorithm. To achieve safe annotations, we use the following procedure: for outputs whose dependence information is completely determined at compile time (i.e., the paths set for the output consists of exactly one path) we omit any the inlet annotations. For each of the other outputs we introduce new labels. The introduction of new labels forces these outputs to be put in separate partitions, thus generating safe partitions.

We could modify the propagation algorithm to capture at least some correlation between outputs. As we provide a better partitioning algorithm (compared to DD) which takes care of such cases, we do not specify the modifications here. One obvious way to utilize the path information as well as retaining the "good" behavior of DD is to utilize the path information to annotate sends and receives only at recursive procedure calls.

The formal method of labeling a call-site is given in the following theorem. As we provide a better partitioning algorithm with a proof of correctness based on operational and denotational semantics in the next chapter, we do not supply a proof for the theorem.

Theorem 3.4 (Annotating Call Sites) *Let f be an n-argument m-result function block and*

$$P(f(x_1, x_2, \ldots, x_n)) = P_f = \{(p_{11}, \ldots, p_{1m}), \ldots, (p_{r1}, \ldots, p_{rm})\}.$$

Then f can be safely approximated by a program graph
$f_g = (\{i_1, i_2, \ldots, i_n, o_1, o_2, \ldots, o_m\}, E(f_g))$ *where,*

$$\begin{aligned} inlet(i_j) &= \phi, \forall 1 \leq j \leq n \\ outlet(o_k) &= \phi, \forall 1 \leq k \leq m \\ E(f_g) &= \{(i_j, o_k) | \forall p \in (P_f \downarrow k), x_j \in p\} \end{aligned}$$

Let $\{a_{11}, \ldots, a_{1m}, \ldots, a_{rm}\}$ be new outlet annotations.

$$\begin{aligned} a_{ks} \in outlet(i_j) \quad if \quad & (i_j, o_k) \notin E(f_g) and \\ & p_s = (p_{s1}, \ldots, p_{sm}) \in P_f \ and \ x_j \in p_{sk} \end{aligned}$$

Let $\{l_1, \ldots, l_m\}$ be new inlet annotations.

$$\begin{aligned} inlet(o_k) &= \phi \ if \ |(P_f \downarrow k)| = 1 \\ &= \{l_k\}, otherwise. \end{aligned}$$

Consider the following example (mentioned in chapter 1).

```
f(x,y,z)=
{a = if x then {b = y + z in b} else { in 2}
 in
   a}
```

$$P(f(x,y,z)) = \{[x,y,z],[x]\}$$

We see from the example above that a dependence edge is added in the program graph

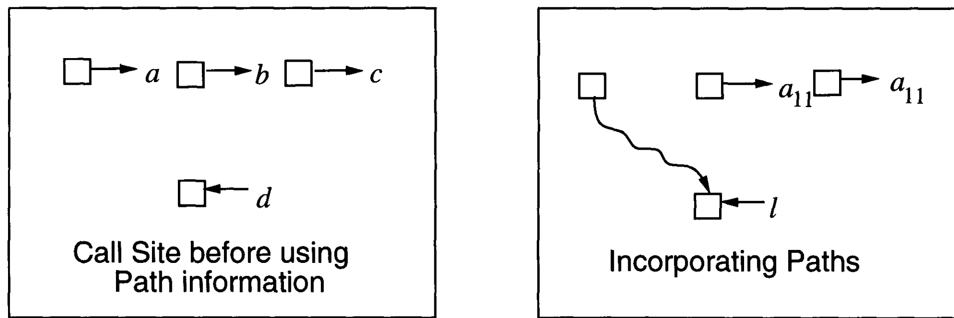


Figure 3.3: Example of Paths used for DD

between the first argument and the result. As this argument is present in all the paths of the result, we can conclude that the result is *strict* in the first argument, and it follows that the addition of the dependence edge is correct. The outlet annotations on the other two arguments sends reflect that fact that the arguments are always used "together", (i.e. if one of them is needed for the result so is the other). This annotation enables the two argument sends to be put in the same thread, even though the function is not strict in either of them. Thus, we are able to make use of the additional information present in paths which is not present in strictness analysis. Using strictness analysis we could have concluded that the dependence edge could be added, but we would not be able to produce the partition mentioned above.

In spite of the theorem, we still may not get good partitions. This may be due to several reasons:

1. The inlet and outlet representation of functions is very brittle. That is, information is lost while being propagated across function boundaries. Consider:

Example 3.4:

```
f(x,y) =
{p = eq? x 0;
 q = y + 1;
 r = y - 1;
 a = g(p,q,r)
 in
   a}
g(p,q,r) =
{a= if p  then { in q}  else { in r}
 in
   a}
```

The DD algorithm does not put the inlets for receiving values of x and y in the same partition. However, the algorithm succeeds in putting x and y in the same partition if g is expanded out in the body of f.

2. Path analysis is expensive to compute - and the algorithm is using only a part of the information to do the partitioning. We probably could do better if the algorithm was directly based on the paths. Consider:

Example 3.5:

```
f(x,y) =
{p = eq? y 0;
 (x1,x2) =  if p  then { in (x,1)}  else { in (1,x)}
 in
   (x1,x2)}


g(a,b,c) =
{(a1,a2) = f(a,c);
 r1 = a1 + a2;
 r2 = r1 + b
 in
   r2}
```

Using path analysis it is possible to detect that $g_p(a,b,c) = \{[a,b,c]\}$, implying that a,b,c can be put in the same partition. If we use theorem 3.4 to introduce path information of f in g, the DD algorithm will not succeed in producing this partition (it will produce the partitions $\{b,c\},\{a\}$).

3. The DD algorithm is inherently limited in representing some kinds of dependence information. Consider:

Example 3.6:

```
f(a,b,c,d,p,q) =
{t1 = a + b;
 t2 = b + c;
 t3 = d + 1;
 t4 = d - 1;
 t5 = g(p,t1,t2);
 t6 = g(q,t3,t4);
 t7 = t5 + t6
 in
   t7}
g(p,q,r) =
{a = if p then { in q} else { in r}
 in
   a}
```

Even if we expand the calls to g in the function f, the DD algorithm fails to detect the dependence of $t7$ on b, and does not put b and d in the same partition. Using the paths of the variables (given in the table below) we detect that the output $t7$ is strict in both b and d.

Var	*Paths*
t1	$[a, b]$
t2	$[b, c]$
t3	$[d]$
t4	$[d]$
t5	$[p, a, b], [p, b, c]$
t6	$[p, d]$
t7	$[p, a, b, d], [p, b, c, d]$

3.4 Summary

In this chapter, we have introduced *Paths*, a way of representing dependence information. This representation is able to capture correlations between non-strict input variables of a function. Paths are computed by abstraction of a non-standard semantics. We have extended the DD algorithm to make use of the information contained in paths when dealing with recursive functions. Finally, we have seen that there is much more information contained in paths than the information exploited by the DD algorithm.

Chapter 4

Partitioning: A New Approach

Previous work on partitioning [37] used a definition of partitioning based on "control", i.e., partitioning was defined as the process of compiling a non-strict program into "threads". Here, we take a different approach. We define partitioning as a transformation from a *source* block to a *destination* block. Thus, we view partitioning as a compiler optimization. The advantage of using this definition is that we can give a formal proof of the correctness of our algorithm based on operational and denotational semantics. Our proof techniques are similar to the ones used in [4] to prove correctness of compiler optimizations.

The rest of the chapter is organized as follows. First, we give a formal definition of the partitioning transformation. Next, we give some preliminary algorithms which are required to define the partitioning algorithm. Then, we provide a new partitioning algorithm for strict blocks and prove its correctness. We extend the algorithm to partition blocks which use non-strictness. In the appendix we study the complexity of the problem: optimal partitioning (producing a partitioning with minimum number of partitions) turns out to be NP-hard.

4.1 Partitioning as transformation

In this section we give the *operational* meaning to the term partitioning. The goal is to capture the intuition behind the execution of threads by block transformation. The threads should display the following behavior:

1. All the instructions in the thread can execute once the first instruction is able to execute.

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

In addition, we should be able to produce a compile time ordering of the instructions in a thread.

We now define the partitions formally:

Definition 4.1 (Partitions) *A partitioning of a block b is a partitioning of a subset of $FV(b) \cup BV(b)$. A variable not present in any partition is assumed to be present in a partition containing only that variable.*

Given a block and a partitioning of it, we aim to produce another block which contains threads, each thread satisfying the requirements given above.

First, we need to transform the given block (b) to another block (b') to introduce explicit receive statements for free variables of the block and values returned by functions. Note that we will not introduce send statements; we assume that such statements are already present in the block b. We also convert a partitioning of b to a partitioning of b'.
Given a block b, we convert it to a block b' as follows:

1. R (for "receives") is a set of variables defined as follows: a variable $x \in R$ if and only if either
   (a) $x \in FV(b)$, or
   (b) There is a statement $(\ldots, \mathtt{x}, \ldots) = \mathtt{e}$ in b, such that $\mathtt{e}$ is not a (strict) primitive operator.

   Let $R = \{x_1, \ldots, x_n\}$. Let $\{x'_1, \ldots, x'_n\}$ be a set of "new" variables (variables not in $FV(b) \cup BV(b)$). There are n statements in b' of the form $\mathtt{x'_i} = \mathtt{x_i}$, for $1 \leq i \leq n$.

2. Define a substitution $\varphi = [x'_1/x_1, \ldots, x'_n/x_n]$. For every statement $st \in b$, there is a statement $st' = \varphi(st)$ in b'.

3. If b returns $(se_1, \ldots, se_n)$ then b' returns $(\varphi(se_1), \ldots, \varphi(se_n))$.

Given a partitioning Q of b we obtain a partitioning $Q' = \varphi(Q)$ of b'.

Example 4.1:

```
b =
 {a = x + y;
  b = y + 1
  in
    (a,b)}
```

with partitions {a,x} and {b,y}. Then, after introducing new (receive) variables c and d we get,

```
b' =
{c = x;
 d = y;
 a = c + d;
 b = d + 1
 in
  (a,b)}
```

with partitions {a,c} and {b,d}.

Observe that the transformation given above enforces the partitions to have the property that variables returned by non-strict expressions are not put in any partition. Only receives of values from these expressions can be put in the same partition.

To facilitate our partitioning transformation, we introduce a family of strict identity operators $S_n, 1 \leq n \leq \infty$ with the following property:

S_n takes n arguments and returns n results. If one of the arguments is undefined, all of the results are undefined. When all of the arguments are defined, S_n has the same effect as n identity functions on the arguments (i.e., S_n returns the arguments as its results). Formally, S_n is described by the rewrite rule:

$$\mathtt{S_n(v_1, \ldots, v_n)} \longrightarrow \mathtt{(v_1, \ldots, v_n)}$$

where each v_i is a ground value. An S_n statement performs a *synchronization* depending on its input values.

We define the meaning of a partitioning using S statements to enforce the behavior of a thread. As our partitioning algorithm works in stages there may be S statements in the block introduced by the "older" stages in the algorithm. However, the meaning of a partition is defined in terms of the original block – the block without any S statements. We can incorporate the deletion of the S statements (to restore the original block) in the partitioning transformation itself. For the sake of simplicity, we omit that from the transformation. The formal transformation is given below:

Definition 4.2 (Semantics of Partitioning) *Given a block* $b = \{\{st\}^* \textbf{ in } (se_1, \ldots, se_t)\}$ *and a partitioning* Q*, the* partitioned *block* b_Q *is defined as follows:*

1. *For each variable* $y \in (FV(b) \cup BV(b))$ *define a unique (new) variable via a mapping* New *as follows.*

   $New(y) = y$, *if* $y \in FV(b)$,

   $New(y) = \underline{y}$, *if* $y \in BV(b)$.

   Let ψ *be the substitution* $[New(y)/y]$.

2. *For each* $q_i \in Q$*:*

   (a) *Let* $\{y_1, \ldots, y_m\} = q_i$.

   (b) *Let* $st_{i1}, \ldots, st_{ir}$ *be statements in* b *corresponding to* q_i.

   (c) *Let* $FV(q_i) = \{x_1, \ldots, x_n\} = (\bigcup_{1 \leq j \leq r} FV(st_{ij})) \setminus (\bigcup_{1 \leq j \leq r} BV(st_{ij}))$.

   (d) *Define new "local" variables* $\{x_1^i, \ldots, x_n^i\}$ *corresponding to* $\{x_1, \ldots, x_n\}$*. Also, define a substitution* $\varphi = [x_1^i/x_1, \ldots, x_n^i/x_n]$.

   (e) *There are statements* $st'_{i0}, \ldots, st'_{i(r+1)}$ *in* b_Q *where,*

   $st'_{i0} ::= (\mathtt{x}_1^{\mathtt{i}}, \ldots, \mathtt{x}_{\mathtt{n}}^{\mathtt{i}}) = \mathtt{S}_{\mathtt{n}}(New(x_1), \ldots, New(x_n))$

   $st'_{ij} ::= \varphi(st_{ij})$, *for* $1 \leq j \leq r$

   $st'_{i(r+1)} ::= (New(y_1), \ldots, New(y_m)) = \mathtt{S}_{\mathtt{n}}(\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{m}})$.

3. *For a statement* $(y_1, \ldots, y_n) = e$ *not corresponding to any partition, there are two statements in* b_Q*:*

   $(\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{n}}) = \psi(e)$ *and*

   $(New(y_1), \ldots, New(y_n)) = (\mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{n}})$.

4. b_Q *returns* $(\psi(se_1), \ldots, \psi(se_t))$.

We say that the partitioning Q induces *the partitioned block* b_Q.

Discussion:

1. The first step in the transformation is to define a new variable for each variable bound in the block. The threads communicate only through these new variables.

2. Each thread consists of three sets of statements:

(a) An S statement which waits for all the free variables of the block and copies these values into local variables.

(b) Statements which compute the variables defined in the thread.

(c) An S statement which waits for all the variables computed in the thread to be defined. Then, the computed variables are copied into the "new" variables.

3. For a statement not corresponding to any partition, we introduce a statement which copies the values computed into the new variables.

Note that we have not ensured that the instructions in a partition have a compile time ordering. However, as our algorithms work only on acyclic blocks, we can produce a compile time ordering by a topological sort of the instructions in the partitions. Observe that if we rewrite every S expression in the partitioned block replacing new variables (introduced in the transformation) by the original variables, we obtain the original block.
For example 4.1, the transformation is given below.

Example 4.2:

```
bQ =
 { (x1,d1) = S2(x,d);
   c = x1;
   a = c + d1;
  (c,a) =  S2(c,a);
   y2 =  S1(y);
   d = y2;
   b = d + 1;
  (d,b) = S2(d,b)
 in
   (a,b)}
```

4.1.1 Correctness of a Partitioning

In this section, we define the notion of correctness used to prove our algorithms. It is similar to the definition of a (partially) correct transformation in [4]. First, we formally define the term *deadlock*.

Definition 4.3 (Deadlock) *Given a term M and a program Pr, we say that M leads to a deadlock if*

1. M has a normal form N, and

2. N is of the form <improper-termination,_>.

The formal definition of a "correct" partitioning is given below. We are essentially saying that if no user defined context can "distinguish" between a partitioned block and the original block, then the partitioned block is correct.

Definition 4.4 (Semantic Correctness of Partitioning) *Given an block b and its partitioned version b_Q, we say that b_Q is correct if for all user definable contexts $C[]$, if $Ans(C[b])$ =<proper-termination,v> then $Ans(C[b_Q])$ =<proper-termination,v>. A partitioning Q is correct partitioning of b if it transforms b into a correct block b_Q.*

In many places, we have mentioned that we are trying to avoid "cycles" in the dataflow graph to obtain correct partitions. That is, incorrect partitioning can at most cause deadlock and cannot affect the computation in any other manner (like changing the value of the answer, for example). In the next theorem, we formalize this intuition by showing that the notion of deadlock and correctness of a partition are closely related. Informally, it states that an incorrect partitioning can only affect the answer by causing a deadlock (in some context).

Theorem 4.1 (Deadlock and correctness) *Given a block b and its partitioned version b_Q, b_Q is incorrect iff $\exists C[]$ such that $Ans(C[b])$ =<proper-termination,v> and $Ans(C[b_Q])$ =<improper-termination,_>.*

Proof: The "if" part of the theorem follows from the definition of correctness. We now prove the "only if" part.

Let $C[]$ be a context which makes b_Q incorrect. Then, we have $Ans(C[b])$ =*<proper-termination,v>* for some v. There are two possibilities for $Ans(C[b_Q])$: either it is *<proper-termination,v'>* for some $v' \neq v$ or *<improper-termination,_ >*. We show that the first case is not possible. If $C[b_Q]$ terminates properly, we know that all the S statements in b_Q have been rewritten. But, the block resulting by rewriting all the S statements is the original block b and hence, it cannot produce a different value. □

4.1.2 Summary

In this section, we have formally defined the meaning of a partitioning by treating it as a block to block transformation. We have defined the notion of correctness of a partitioning and shown that it is closely related to whether the block after partitioning deadlocks.

4.2 Partitioning: Preliminary Algorithms

In this section, we give two algorithms which are needed for partitioning. The first algorithm is an algorithm to do subpartitioning. This is similar to the subpartitioning algorithm used by the DD algorithm. The second algorithm converts a block with cyclic dependencies to a block which is acyclic. This is required as our partitioning algorithms work only with acyclic blocks. The following definition formalize the notion of an acyclic block.

Definition 4.5 (Dependence graph) *A* dependence graph *of a block b is the directed graph $G_{db} = (V_{db}, E_{db})$ where*
$V_{db} = FV(b) \bigcup BV(b)$, *and*
$(x, y_i) \in E_{db}$ *if there is a statement* $(y_1, \ldots, y_i, \ldots, y_n) = e$ *and* $x \in (P(e) \downarrow i)$. *If e is neither a simple expression nor a primitive operator then (x, y_i) is assigned the weight* 1.
We call b an acyclic block if G_{db} does not contain any cycles.

The dependence graph represents the various (certain and potential) dependencies in the block. The way we detect dependencies is by looking at the paths for the expression; there is a dependency if a variable is present in one of the paths. For convenience, we say there is a path from x to y in b, when we mean that there is path from x to y in the dependence graph of b. Variables x and y are connected to each other in b iff there is a path from x to y in b or from y to x in b. Instead of using squiggly edges to break threads across function calls like DD does, we use weighted edges instead.

For example 1.2, the dependence graph is given below.

4.2.1 Subpartitioning

As we cannot (in general) execute non-strict expressions in a single thread, we need a method to safely *subpartition* the partitions obtained. The subpartitioning algorithm is:

Algorithm S

Given a partitioning Q, for each q in Q do:
Subpartition q into $q_1, \ldots, q_n$ such that every $\forall i, 1 \leq i \leq n, \exists c, \forall x \in q_i, depth(x) = c$ in G_{db}. That is, all variables which are "equally deep" in the (weighted) graph are put in the same partition.

Subpartitioning ensures that the following properties are satisfied.

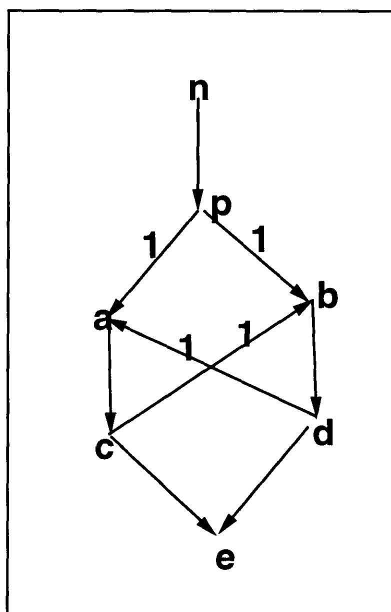


Figure 4.1: A Dependence Graph

1. The partitions themselves should not introduce a "cycle" in the block. This could happen in the following scenario: there are two instructions in a thread $T1$ such that one depends on another through an instruction in a different thread $T2$. As we require that a thread should completely execute before returning any value, this will result in a deadlock.

2. Subpartitioning "breaks" the thread at every call to a non-strict operator or a function call. This is in accordance with the choice we have made. If (say) we choose to execute (strict) function calls in the same thread, we could change the weights in the dependence graph to give the desired result.

We prove the following lemma about subpartitioning.

Lemma 4.2 (Safe Subpartitioning) *Let q be a partition in a partitioning Q of b. Let q_1, q_2 be subpartitions of q such that $FV(q_1) \subseteq FV(q)$. If Q is a correct partitioning, so is $Q' = (Q \setminus \{q\}) \bigcup \{q_1, q_2\}$.*

Proof: We need to show that if all the statements in the block b_Q induced by Q execute (i.e., get rewritten), so do all of the statements in the block $b_{Q'}$ induced by Q'. The only difference between b_Q and $b_{Q'}$ is that the statements corresponding to the partition q

are replaced by those corresponding to the partitions q_1 and q_2.

Observe that the statements in the partition will execute only if the first S statement in the partition executes. This implies that variables in $FV(q)$ are all bound to a value when the partition corresponding to q starts executing. As, $FV(q_1) \subseteq FV(q)$, the first S statement in q_1 partition can execute, allowing the rest of the statements in the q_1 partition to be executed.

Finally, note that the q_2 partition can execute, as $FV(q_2) \subseteq (FV(q) \bigcup q_1)$. □

The proof of correctness of subpartitioning is given separately. The proof uses the properties of the partitioning algorithm given later. The informal reason why we cannot use the lemma to prove correctness is that the lemma expects a correct partitioning to start with. This property will not (in general) be true of the partitions we provide to the subpartitioning algorithm. The subpartitioning algorithm makes these possibly incorrect partitions correct.

4.2.2 Converting cyclic to acyclic blocks

We convert a cyclic block to an acyclic block such that the partitions of the acyclic block can be converted back to partitions of the original block. Our reasons for choosing an indirect approach are these: first, we would like the partitions themselves not to contain internal cycles. This enables us to give a compile time ordering by a topological sort of the instructions. Second, our algorithm to do partitioning is similar to DD in the sense that we propagate labels through a block. Having cycles in the block raises difficult questions on how to propagate labels around a cycle and when to stop propagating labels. We eliminate these problems by working on an acyclic block.

To convert a block with cycles to an acyclic block, we use the standard technique of removing a set of vertices from the dependence graph. The DD algorithm works similarly: it "breaks" basic blocks at every function call or conditional statement. This is equivalent to removing a set of vertices from the dependence graph to make it acyclic. However, DD is conservative in its approach in the sense that it removes all vertices which could give rise to cycles in the block, whereas we look for cycles in the graph and remove only those vertices which are part of a cycle. In this manner, we are able to generate a "bigger" acyclic block and analyze blocks at a bigger granularity, which should enable us to generate better partitions. In some sense, our scheme "taxes" the programmer only when he/she

uses non-strictness to feed results back to arguments.

There is a price in going through acyclic blocks: we essentially "move" the cycles inside the block to cycles outside the block (completed through input and output variables). This leads to the possibility of generating many more cycles (other than the cycles present in the original graph) by arbitrary feedback between outputs and inputs. Thus, our partitioning algorithm is more conservative than it should be.

Definition 4.6 (Feedback set) *A* feedback vertex set *of a directed graph G is a subset (S) of vertices of G such that the graph G' obtained by removing vertices in S (and their incident edges) is acyclic. A* minimal *feedback vertex set is a feedback vertex set such that no proper subset of it is a feedback vertex set.*

Note: Though computing the *minimum* feedback vertex set is NP-hard [14], computing a *minimal* set can be accomplished by Depth First Search [12].
The algorithm to convert a block with cycles to an acyclic block is given below.
Algorithm C
Given an arbitrary block $b = \{st^* \textbf{ in } (se_1, se_2, \ldots, se_n)\}$, we partition it as follows:

1. G_{db} is the dependence graph of b.

2. $S = \{y_1, \ldots, y_m\}$ = A *minimal* feedback vertex set of G_{db}.

3. $S' = \{z_1, \ldots, z_m\}$ = A set of new variables such that $\forall i, z_i \notin (FV(b) \bigcup BV(b))$. Define the substitution $\varphi = [z_1/y_1, \ldots, z_m/y_m]$.

4. b' is the block formed by:

   (a) For every statement $st \in b$ there is a statement $st' = \varphi(st)$ in b'.

   (b) $b' = \{st^* \textbf{ in } (se_1, \ldots, se_n, y_1, \ldots, y_m)\}$.

5. Let $\{q_1, q_2, \ldots, q_k\}$ be a partitioning of the acyclic block b'. Then the partitions of b are $\{q_i'\}$ where $q_i' = q_i \setminus \{z_1, \ldots, z_m\}$ for $1 \leq i \leq k$.

Consider example 1.2. For that block, (refer to Fig 4.1) a possible minimal feedback vertex set is $\{a, b\}$. Then, b' (with new variables g and h) is:

Example 4.3:

```
f(n)=
 {p = eq? n 0;
  a =  if p  then { in 3}  else { in d};
  b =  if p  then { in c}  else { in 4};
  c = g + 2;
  d = h + 1;
  e = c * d
 in
   (e,a,b)}
```

If (say) we obtain the partitions $\{p, n\}$, $\{g, c\}$, $\{h, d\}$, $\{e\}$, $\{a\}$, $\{b\}$ for the block b', then algorithm C generates $\{p, n\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a\}$, $\{b\}$ for the block b.

We now give the correctness proof for algorithm C.

Theorem 4.3 (Correctness of algorithm C) *If the partitioning of the block b' in algorithm C is correct, then the partitioning of the block b is correct.*

Proof: Let $Q = \{q_1, \ldots, q_n\}$ be a correct partitioning of b'. First, we show that input variables can be safely put in a separate partition by themselves. If q_i contains one of the new variables (say z_j) introduced by the algorithm, then it can be split into two partitions $q_{i1} = q \setminus \{z_j\}$ and $q_{i2} = \{z_j\}$. The safety of the new partitions is guaranteed by lemma 4.2 as $FV(q_{i2}) = \{z_j\}$ and $z_j \in FV(q_i)$, as z_j is an input variable. Proceeding in this manner, we generate the correct partitioning $Q' = \{q'_1, \ldots, q'_n, \{z_1\}, \ldots, \{z_m\}\}$.

Let C be a context for b. Without loss of generality, we can assume that the context is of the form:

{....

$(\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_\mathrm{n}) = b;$

....}

The context is equivalent to a context C' for b' of the form:

{....

$(\mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_\mathrm{n}, \mathrm{z}_1, \ldots, \mathrm{z}_\mathrm{m}) = b';$

....}

This is due to the fact that replacing "uses" of z_i's by y_i's and eliminating the z variables yields the block b.

By correctness of Q', we can replace b' by $b'_{Q'}$, the partitioned version of b'. Now, eliminating the z variables again gives the context C with the partitioned version of b, thus the partitioning is correct. □

Note: The proof of the theorem does not depend on the fact that the variables $\{y_1, \ldots, y_m\}$ form a (minimal) feedback vertex set. Also, algorithm C has to "drop" n variables among $\{y_1, \ldots, y_m, z_1, \ldots, z_m\}$ from the partitions. The version given in the algorithm removes the variables $\{z_1, \ldots, z_m\}$. We could employ a heuristic here which removes $\{y_1, \ldots, y_m\}$ if doing so leads to bigger partitions. For the example given, a better partitioning of b would be the partitions $\{p, n\}$, $\{a, c\}$, $\{b, d\}$, $\{e\}$, which can be obtained by using the heuristic.

4.2.3 Summary

In this section, we introduced two preliminary algorithms required for partitioning. One of them is the algorithm to do subpartitoning, which ensures that non-strict expressions are put in separate threads. We have to be careful that we do not introduce new cycles while we are doing subpartitioning. Though we prove this fact later, informally it is achieved by splitting the partition into subpartitions of equal "depth" in the dependence graph.

The second algorithm converts blocks with cyclic dependencies to an acyclic block. We can now deal with only such blocks, considerably simplifying the partitioning algorithms given later. This algorithm works by finding a minimal feedback vertex set of the dependence graph of a block and "removing" variables corresponding to these vertices from the block.

4.3 Partitioning strict blocks

In this section, we consider only acyclic blocks which are formed by strict primitive operators. We give an algorithm which partitions such blocks.

Algorithm AB

Given an acyclic block $b = \{\{st\}^* \textbf{ in } (se_1, \ldots, se_m)\}$:

1. Let G_{db} be the dependence graph of b.

2. Define $Var = FV(b) \cup BV(b)$. For each $y \in Var$, find two sets of labels $A(y)$ and $B(y)$ as follows:

   (a) (Backward pass)

   For output se_j, $A(se_j) = \{j\}$ (for $1 \leq j \leq m$).

   For any other variable y, $A(y) = \bigcup_{(y,x) \in G_{db}} A(x)$.

(b) (Forward pass)

If $x \notin BV(b)$, $B(x) = A(x)$.

For any other variable y, $B(y) = \bigcap_{(x,y) \in G_{db}} B(x)$.

3. Do the following until the partitions do not change.

   (a) All variables with same A's are grouped in the same partition. Given a partition q, force $B(y)_{y \in q} = \cap_{z \in q} B(z)$. Propagate changes to B's in a forward pass.

   (b) All variables with same B's are grouped in the same partition. Given a partition q force $A(y)_{y \in q} = \cup_{z \in q} A(z)$. Propagate changes to A's in a backward pass.

The A sets computed by the algorithm are exactly the same as demand sets of the DD algorithm. That is, the A set of a variable represents the set of outputs which depend upon the variable. Thus, the A stage of the partitioning algorithm (when variables with same A's are put in the same partition) is the same as the demand stage in the DD algorithm. The B sets of a variable x are those outputs which *can* be made to depend on x *without* affecting the input-output behavior. What can we put in $B(x)$? Clearly, if x is an input variable, we can put the outputs in $A(x)$. These dependencies already exist in the block. Also, we cannot include any other output in the $B(x)$ set; it would make that output depend on x, introducing a dependency in the block which did not exist previously. If x is not an input variable, then it has a set of *parents* $\{y_1, \ldots, y_n\}$ in G_{db} (that is, $(y_i, x) \in E(G_{db})$). Any variable which is present in $B(y_i)$ for $1 \leq i \leq n$ can be put in $B(x)$ because adding this dependency is still safe.

Note that we have not used subpartitioning in the algorithm given. This is to be expected as algorithm S has an effect only on blocks containing non-strict operators. An example of the B stage of the algorithm is given in figure 4.2. The algorithm is shown operating on the dependence graph. For clarity, we have used characters (a,b,...) instead of numbers to denote outputs. For a vertex in the dependence graph, the characters at the lower left corner specify its A set and the ones at the upper right corner specify its B set. Note that we have obtained better partitions than the iterated DD partitioning given in chapter 1.

Our strategy for proving correctness for the AB algorithm is this: first, we define a new correctness criterion for partitioning strict blocks. This criterion operates on the structure of dependencies in the block, rather than on operational semantics. Then, we show that this criterion is sufficient to ensure the correctness with respect to the semantic criterion. Finally,

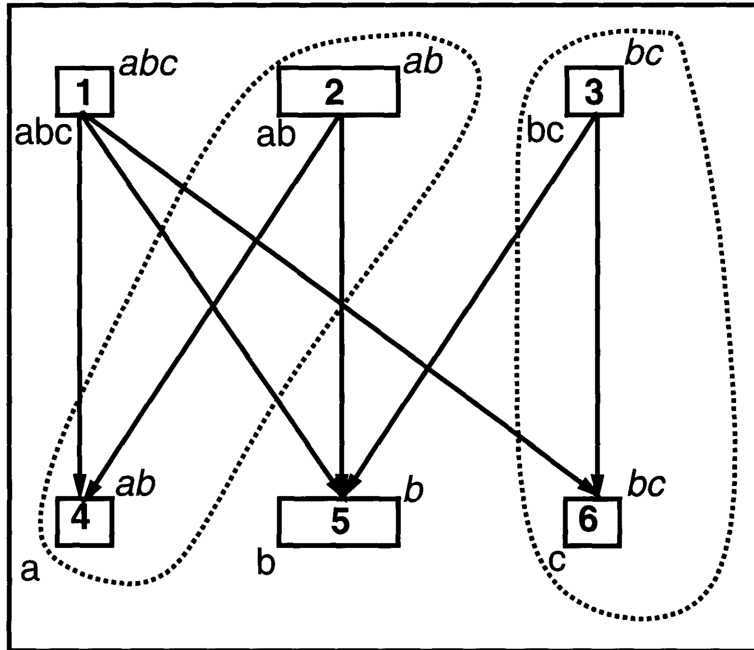


Figure 4.2: B stage of algorithm AB

we prove the correctness of algorithm AB using the dependence graph based correctness criterion.

4.3.1 Dependence graph based Correctness of a partitioning

Informally, we are trying to ensure the following: the input-output behavior of a block *must not* be altered by partitioning; however, we are free to make any internal changes to the block. For strict blocks, input-output behavior is characterized by the various dependencies that exist between input and outputs, which show up as connections in the dependence graph. Thus, our correctness criterion states that the input-output connectivity should be unchanged by the partitioning algorithm. Formally,

Definition 4.7 (Dependence graph correctness) *Given an acyclic block b and its partitioned version b_Q, we say that b_Q is* correct *of b if the following conditions hold:*

1. *(acyclic) The dependence graph of b_Q is acyclic.*
2. *(connectivity) Let $\{x_1, \ldots, x_n\} = FV(b) = FV(b_Q)$, and $y_1, \ldots, y_m$ be* outputs *of the blocks. Then, x_i is connected to y_j in b_Q iff x_i is connected to y_j in b.*

A partitioning $Q = \{q_1, \ldots, q_r\}$ of block b is correct if it generates a correct partitioned version b_Q of b.

4.3.2 Safety of the Dependence graph Criterion

Here, we prove that the correctness criterion based on the dependence graph is sufficient to guarantee the correctness criterion for blocks consisting of strict primitive operators. We need the following lemma which relates a characteristic of the dependence graph (i.e., it possessing a cycle) to a property of the term (i.e., the term leading to deadlock).

Lemma 4.4 (Characterizing deadlocks) *A term M of SP-TAC leads to deadlock iff the dependence graph of M contains a cycle.*

Proof: 1. ($\Leftarrow$) Observe that for primitive operators, a precondition for any rewrite rule to be applicable is that the arguments of the primitive operators must all be values. If we assume that any statement in a cycle of M gets rewritten, we derive a contradiction that the variable in the statement must have already been a value.

2. ($\Rightarrow$) Assume that the normal form of M is the term N, which is a deadlock. From the rewrite rules, we can conclude that M contains a cycle iff N does. As N is a deadlock, N is of the form $\{st_1; st_2; \ldots \textbf{in}\ _\}$. Consider the dependence graph of N. Every vertex in this graph should have an edge pointing to it, or a rewrite rule would be applicable and N would not be in normal form. This proves that the dependence graph of N contains a cycle, as an acyclic graph always has at least one vertex which does not have any incoming edges. Thus, the dependence graph of M contains a cycle.

□

Theorem 4.5 (Sufficiency of dependence graph criterion) *Let b be a block and b_Q its partitioned version. If b_Q is correct with respect to the dependence graph criterion, the b_Q is correct with respect to the semantic criterion.*

Proof: We prove the theorem by contradiction. Assume that b_Q is not correct with respect to the semantic criterion. That is, $\exists C, Ans(C[b]) \neq Ans(C[b_Q])$. Due to the property of partitioning (theorem 4.1), this can only happen if $C[b]$ leads to a proper termination and $C[b_Q]$ leads to a deadlock. By lemma 4.4, the dependence graph of $C[b]$ must contain

be acyclic and the dependence graph of $C[b_Q]$ must contain a cycle (say CY). We derive a contradiction in all the possible cases.

1. CY consists of vertices only from C: Thus, $C[b]$ contains CY too, a contradiction.
2. CY consists of vertices only from b_Q: Contradicts the acyclic condition of the dependence graph criterion.
3. CY consists of vertices from b_Q and C: Let $CY = CY_1, CY_2, \ldots, CY_n$, where each CY_i is a set of consecutive vertices in CY and each CY_i is contained entirely within b_Q or C. Let CY_k be one set contained entirely within b_Q. Note that CY_k forms a path in b_Q from an input vertex to an output vertex. Thus, by the connectivity condition, there is a path CY'_k in b between the same input and output vertices present in CY_k. We can now construct a cycle CY' in $C[b]$ using CY'_k instead of CY_k (for such k), a contradiction.

□

4.3.3 Strict blocks: Correctness of the AB algorithm

The following properties are true of A and B sets computed for strict blocks:

Lemma 4.6 *For a variable x, $A(x) \subseteq B(x)$.*

Proof: The proof is by induction on the structure of the block b.

Basis: For input variables, the lemma is trivially satisfied.

Induction: For any other variable y, $B(y) = \cap_{(x,y)\in G_{db}} B(x)$. We show that if an output (numbered j) is in $A(y)$ then, $j \in B(y)$. By the definition of A sets, $j \in A(x)$ for any x such that $(x, y) \in G_{db}$. By induction hypothesis, $A(x) \subseteq B(x)$ for all such x. Thus, $j \in B(x)$ for such x, implying that $j \in B(y)$. □

Theorem 4.7 (Subset Relations in Strict Blocks) *If there is a directed path from a variable x to a variable y in a block b, then*

1. $A(y) \subseteq A(x)$.
2. $B(y) \subseteq B(x)$.

Proof: An easy induction on the length of the path. □

The following theorem relates labels and the connectivity condition. It says that two variable can be put in the same partition if they can "tolerate" each other. By "tolerate" we mean that the outputs dependent on one variable (the A set) should be present in the B set of the other variable (and vice-versa).

Theorem 4.8 (Necessary and sufficient conditions for connectivity) *Let b be a block such that A and B sets are computed for every variable in b. Let $q \subseteq (FV(b) \bigcup BV(b))$ and $Q = \{q\}$ a partitioning of b. Then, Q satisfies the connectivity condition if and only if $\forall y_i, y_j \in q$*

$$(A(y_i) \subseteq B(y_j)) \bigwedge (A(y_j) \subseteq B(y_i)).$$

Proof: For a variable y let $I(y)$ denote the set of inputs (i.e. the free variables of b) connected to y in b. We make the following observations:

1. There is a path from an input x to an output z in the partitioned block b_Q if and only if there exist y_i and y_j in the same partition, such that there is a path from x to y_i in b and from y_j to z in b. If $y_i = y_j$, we obtain paths originally in b.
2. An output z_r is connected to y in b if and only if $r \in A(y)$.
3. An output z_r is connected to every input in $I(y)$ if and only if $r \in B(y)$.

The first property follows from the partitioning transformation. The properties of A and B sets follow by an easy induction on the computation of A and B sets.

We now proceed to prove the theorem.

1. ($\Longrightarrow$) Assume that one of the conditions is not satisfied. Then

$$\exists r \in A(y_i), r \notin B(y_j)$$

   We claim that the output z_r is not connected in b to at least one of the inputs (say x_l) in $I(y_j)$ (by observation 3). Thus, after partitioning we introduce a path between x_l and z_r in b_Q, and hence b_Q is not correct.

2. ($\Longleftarrow$) Assume that the partition is not correct. That is, there is an input x connected to z_r in b_Q but not in b. By observation 1, there are $y_i, y_j \in q$ such that x is connected to y_i and y_j is connected to z_r. Therefore, $r \in A(y_j)$. As there is no path from x to z_r in b, $r \notin B(y_i)$ (by observation 3). Thus $A(y_j) \not\subseteq B(y_i)$.

□

Corollary 4.8.1 *If a partition q satisfies the conditions of theorem 4.8, then the A sets (and hence, B sets) of input variables of b are unchanged in b_Q the partitioned block induced by $Q = \{q\}$.*

Proof: This follows directly from observation 2 in the proof of theorem 4.8 which says that the A sets of the input variables capture all possible input-output connections in a block, which are unchanged by theorem 4.8. □

In spite of the fact that the condition of theorem 4.8 is necessary and sufficient, we do not use it in partitioning because the condition is not transitive: if we conclude i, j can be put in the same partition and j, k can be put in the same partition, it does not follow that i, j, k can all be together in a partition. Thus, we need to use some sort of graph coloring technique to obtain partitions. We formalize this in the appendix, by proving that obtaining optimal partitions is NP-hard, i.e., we cannot do better than coloring if we use the necessary and sufficient condition.

We now give the correctness proof of the algorithm AB. Basically, the fact that we can safely put variables with the same A's (or B's) together in the same partition follows directly from the connectivity theorem above. However, we need to prove additional facts to show that the entire algorithm (working in stages) is correct.

Theorem 4.9 (Correctness of AB on strict blocks) *Algorithm AB correctly partitions a block b.*

Proof: First, note that the two conditions (A and B) are enough to guarantee that conditions of theorem 4.8 are satisfied (follows from lemma 4.6).

We can think of the algorithm working in stages: generating partitions, applying the partitions obtained to get the partitioned block, and repeated partitioning of this block. We show that after a stage:

1. Partitions are correct: We first show correctness for the A stage. The problem is that theorem 4.8 guarantees correctness for forming a *single* partition, whereas the A stage could form several partitions simultaneously. However, we can easily see that the change affects only the B labels, and the partitions formed are still valid. During the B stage the following could happen: after propagating changes to the A sets, we discover that some of the B sets have changed (as the B sets depend on

the A sets). This could make the partitions we have formed "invalid". However, we know that (corollary 4.8.1) for any input the B set remain unchanged. Thus, by the computation of B sets, the B set of any other variable is also unchanged. Acyclicity is ensured by the A and B stages themselves. We show this for the A stage. Assume there is a cycle formed by partitions $q_1, \ldots, q_n$. Let $A(q)$ stand for the A set of all variables in q. As there is a path between a variable in q_1 to a variable in q_2, $A(q_1) \supseteq A(q_2)$. Extending this argument to successive partitions in the cycle we have $A(q_1) \supseteq A(q_2) \ldots \supseteq A(q_n) \supseteq A(q_1)$, implies that all the A's for these partitions must be same and the algorithm would have combined $q_1, \ldots, q_n$ into a single partition. A similar argument can be made for the B stage.

2. New labels reflect the changes in block: easy.

3. Partitions are extended (i.e. the old partitions are not broken up.): This follows from the fact that if two variables are put in the same partition, their A and B sets will be identical, ensuring that they will be present in the same partition at every subsequent stage.

Using these observations, the correctness of the entire algorithm follows by an easy induction on the number of stages. □

4.3.4 Summary

In this section, we have dealt only with strict blocks. First, we presented an algorithm to partition them. The algorithm uses A sets (which are similar to demand sets of the DD algorithm) and B sets which are computed from A sets. We have proved the correctness of the algorithm by studying the structure of the dependence graph and the effects partitioning produces on the structure of the graph. We also have shown that preserving the structure of the dependence graph is sufficient to ensure semantic correctness of the partitioning transformation.

4.4 Partitioning non-strict blocks

In this section, we extend the partitioning algorithm defined for strict blocks to work on non-strict blocks. The idea is to use the paths of an expression to know the dependence information of expressions which may not be strict. We know that at run-time one of the

paths will be valid for the expression. Thus, we can view the execution of a non-strict block as follows: first, make a *choice* at each non-strict expression regarding the path which is valid at run-time. Then, the execution of the non-strict block works *exactly* like a strict block, which we know how to partition. The algorithm for non-strict blocks works by considering all possible choices of the non-strict statements to (conceptually) generate all the strict blocks which could result due to the execution of the non-strict block.

The algorithm is again based on A and B sets. The only difference is that the outputs are *tagged* by the strict block which they arise from. The tags are computed as follows. Let there be n non-strict operators or functions in the block. For each $1 \leq i \leq n$, let m_i represent the number of paths of the non-strict operator. Then, the total number of strict blocks that can result during the execution of the non-strict block is $m_1 \times \cdots \times m_n$. Each strict block can be uniquely identified with the set of choices made at the paths in each non-strict expression. That is, the tag is a set $\{(i, j_i)\}$ for $1 \leq i \leq n$ and j_i is some number between 1 and m_i corresponding to the j_i^{th} path. The formal definition of labels (i.e., tagged outputs) is given below:

Definition 4.8 (Labels) *A label is of the form* $(m, n, \{(i, j_i)|(1 \leq i \leq n)\})$ *for integers* m, n, i, j_i*'s.*

We now give the algorithm to compute the A and B sets and use them to do partitioning. As the input block is acyclic, we can define an ordering on the variables of the block such that variables occurring at some position in the ordering depend only on variables occurring in lesser positions in the ordering. Like the AB algorithm for strict blocks, the algorithm for non-strict blocks works by doing a series of backward and forward passes on this ordering.

First, we give the formal definition of the variable ordering and of $Before$, the set of variables before a given variable in the ordering.

Definition 4.9 (Ordering of variables) *Given an acyclic block* b*, let* G_{db} *be its dependence graph. Then, the canonical ordering of variables is a sequence (*Ord*) of variables such that*

$$(x, y) \in G_{db} \Longrightarrow Ord(x) < Ord(y).$$

For a variable y_j *defined in a statement* $(y_1, \ldots, y_m) = e$ *of* b*,*

$$Before(y_j) = \{x | Ord(x) < Ord(y)\}$$

The ordering can be computed by a topological sort of the dependence graph G_{db}.

The partitioning algorithm is given below:

Algorithm AB

Given an acyclic block b:

1. Each statement of b which is not a primitive operator is annotated with a unique number from $1 \ldots n$ where n is the number of such statements. Also, the numbering respects the topological ordering of the block. For each such statement $(\ldots) =_i e$ let $P_i = P(e) = \{p_{i1}, \ldots, p_{ik}\}$.

2. Let $L_{(i,r)}$ stand for all labels (n_1, n_2, l) such that $(i, r) \in l$.

3. Define $Var = FV(b) \cup BV(b) \cup \{u\}$, where u is a new variable. Make u the first variable in the ordering of variables.

4. For each $y \in Var$, find two sets of labels $A(y)$ and $B(y)$ as follows:

   (a) (Backward pass)

   If $b = \{st^* \textbf{ in } (se_1, \ldots, se_m)\}$, then $\forall j, A(se_j) = \{(j, n, \{(i, l_i) | 1 \leq i \leq n\}) | 1 \leq l_i \leq |P_i|\}$.

   For a statement $y = e$, where e is a primitive operator,

   $$\alpha \in A(y) \Longrightarrow \forall x \in FV(e), \alpha \in A(x)$$

   For any other statement $(y_1, \ldots, y_t) =_i e$ such that $P(e) = \{p_{i1}, \ldots, p_{ik}\}$, a label $\alpha = (j, n, l) \in A(x))$ if and only if there are r, s such that $\alpha \in A(y_s)$ and $x \in (p_{ir} \downarrow s)$ and $x \in Before(y_s)$ and $(i, r) \in l$.

   (b) (Forward pass)

   If $x \notin BV(b)$, then $B(x) = A(x)$.

   For a statement $y = e$, where e is a primitive operator

   $$B(y) = \bigcap_{x \in FV(e)} B(x)$$

   For any other statement $(y_1, \ldots, y_t) =_i e$ such that $P(e) = \{p_{i1}, \ldots, p_{ik}\}$, $\alpha \in B(y_s)$ if and only if there is an r such that $\alpha = (j, n, l)$ and $(i, r) \in l$ and $\forall x \in ((p_{ir} \downarrow s) \cap Before(y_s))\alpha \in B(x)$. That is,

   $$B(y_s) = \bigcup_{1 \leq r \leq k} ((\bigcap_{x \in ((p_{ir} \downarrow s) \cap Before(y_s))} B(x)) \cap L_{(i,r)})$$

5. As in the AB algorithm for strict blocks, do alternate A and B stages (with subpartitioning) to produce partitions.

Discussion: The variable u in the algorithm stands for the undefined variable i.e., the value of u is always $\perp$. The propagation of labels through primitive operators is done in exactly the same manner as in strict blocks. For a non-strict statement $(y_1, \ldots, y_t) =_i e$ with paths $\{p_{i1}, \ldots, p_{ik}\}$ the propagation is done as follows:

1. Assume we have a label $\alpha \in y_s$ for some $1 \leq s \leq t$. Let α contain a choice of the form (i, r). That is, α is present in the strict block where this statement "makes" the rth choice among the paths. So, we should propagate this label to all variables $x \in (p_{ir} \downarrow s)$. If $(p_{ir} \downarrow s) = \perp_p$ we can safely propagate it to any variable. But, we propagate it to only variables before it, ensuring that labels do not propagate in a circular manner.

2. Computing the B sets is done by a forward pass. We have mentioned that the B set of a variable y_s stands for outputs which can safely be made to depend on y_s. The computation is very similar to the computation of B sets from a primitive operator. The only difference is that we consider each choice of the expression (i.e., which path it takes) and combine the B sets for all the choices. The condition that variable x should be before y_s is required to take care of the case when the path is $\perp_p$.

The following example illustrates the AB algorithm.

Example 4.4:

```
f(x,y)=
  {p = eq? y 0;
   a =1 if p then { in x} else { in 3};
   b = a + 1
 in
   b}
```

The paths set of the non-strict statement in the block is $P_1 = \{[p, x], [p]\}$. In this example, we do not need the undefined variable u as none of the paths are $\perp_p$. There is only one output (b) and only one non-strict statement. Thus, the labels we propagate in this block are of the form $(1, 1, l)$ where l is a singleton set consisting of either $(1, 1)$ or $(1, 2)$ depending on whether the label corresponds to the path $[p, x]$ or $[p]$. As b is an output, it gets all possible labels, that is, $A(b) = \{(1, 1, \{(1, 1)\}), (1, 1, \{(2, 2)\})\}$. Call these labels α and β. The backward propagation of the labels is given in the following table.

Var	*Backward Propagation Stage*			
	Initial	b	a	p
x			α	α
y				α, β
p			α, β	α, β
a		α, β	α, β	α, β
b	α, β	α, β	α, β	α, β

The point to note is that $A(x)$ only has the label α. The label β does not "reach" x as β corresponds to a different path (namely $[p]$). However, both the labels reach p as it is present in both the paths of the non-strict operator. It turns out that the B sets produced are exactly equal to the A sets, and thus they do not add further information. The A stage of the algorithm produces (before subpartitioning) the partitioning $\{x\}, \{y, p, a, b\}$. Subpartitioning (due to the **if** statement) will produce $\{x\}, \{y, p\}, \{a, b\}$.

From the algorithm to compute A and B sets, we have the following property of A and B sets.

Lemma 4.10 *For a variable x, $A(x) \subseteq B(x)$.*

Proof: The proof is by induction on the structure of b.

Basis: For input variables, the result is true.

Induction: For variables computed by primitive operators, the result is true (by the proof of lemma 4.6). Consider any other variable y_s computed in a statement $(y_1, \ldots, y_t) = e_i$. Let $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ik}\}$. We show that if $\alpha \in A(y_s)$ then $\alpha \in B(y_s)$. Let $\alpha = (j, n, l)$ such that $(i, r) \in l$. From the computation of A sets we have, $\forall x \in ((p_{ir} \downarrow s) \cap Before(y_s))$, $\alpha \in A(x)$. By induction hypothesis, $\alpha \in B(x)$ for such x. By computation of B sets, we have $\alpha \in B(y_s)$. □

We have a theorem which is similar to theorem 4.7 for the A and B sets computed here. This follows directly from the theorem corresponding to strict operators as we are considering only paths in the dependence graph formed through primitive operators.

Theorem 4.11 (Subset Relations for Non-strict Blocks) *If there is a (directed) path of weight 0 from a variable x to a variable y in the dependence graph of b, then*

1. *$A(y) \subseteq A(x)$.*

2. $B(y) \subseteq B(x)$.

We now prove the correctness of the AB algorithm working on non-strict blocks. We "simulate" non-strictness by using a family of non strict operators N_{ji} taking ji values as arguments and returning i values.

The meaning of the statement

$(\mathtt{x_1}, \ldots, \mathtt{x_i}) = \mathtt{N_{ji}}(\mathtt{y_{11}}, \mathtt{y_{12}}, \ldots, \mathtt{y_{1i}}, \ldots, \mathtt{y_{ji}})$

is that it could be *rewritten* to any one of the j statements

$\{(\mathtt{x_1}, \ldots, \mathtt{x_i}) = (\mathtt{y_{k1}}, \mathtt{y_{k2}}, \ldots, \mathtt{y_{ki}}), 1 \leq \mathtt{k} \leq \mathtt{j}\}$..

We transform a block with non-strictness to a block containing only strict operators and these N-operators. We prove that the transformation preserves the meaning of the block (in a denotational sense). First we give a method to transform a non-strict expression. The idea is based on using strict operators to compute values corresponding to all the paths of the expression and using a single N-operator to choose any one of the values computed. This reflects the uncertainty at compile time; we do not know which one of the paths will be valid at run time.

Definition 4.10 (N-block equivalence) *Let pr be a program $fenv = \mathcal{E}_p[\![pr]\!]$ and $aenv = \mathcal{A}_p[\![pr]\!]$. For a block b, an equivalent N-block b_N is defined as follows.*

1. *A statement $st ::= y = e$ of b where e is a simple expression or primitive operator is present unchanged in b_N.*

2. *An expression e in any other statement $(y_1, \ldots, y_m) = e$ is replace by the a block b_{Ne} defined as follows.*

   (a) *Let u be a new variable which represents $\perp$. Let $P(e) = \{(p_{i1}, \ldots, p_{im})\}$ for $1 \leq i \leq n$.*

   (b) *For each i, j such that $1 \leq i \leq n$ and $1 \leq j \leq m$, there is a statement defined by:*

      i. *$p_{ij} \neq \perp_p$: Let $p_{ij} = [z_1, \ldots, z_k]$. There is a statement*

$$\mathtt{x_{ij}} = \mathtt{op_{ij}}(\mathtt{z_1}, \ldots, \mathtt{z_k})$$

where op_{ij} is a (strict) operator such that

$op_{ij}(v_1, \ldots, v_k) = \perp$, if one of $\{v_1, \ldots, v_k\}$ is $\perp$.

$op_{ij}(v_1, \ldots, v_k) = \mathcal{E}[\![e]\!]\,[z_l \mapsto v_l]\, fenv$, otherwise.

ii. $p_{ij} = \perp_p$: *let* $\{z_1, \ldots, z_k\} = Before(y_j)$. *There is a statement*

$$\mathtt{x_{ij}} = \mathtt{Strict}(\mathtt{z_1}, \ldots, \mathtt{z_k}, \mathtt{u}).$$

The *Strict* statement "waits" for all its arguments and returns an (arbitrary) constant value as the output.

(c) We have a single N-statement:

$$(\mathtt{y_1}, \ldots, \mathtt{y_m}) = \mathtt{N_{nm}}(\mathtt{x_{11}}, \ldots, \mathtt{x_{1m}}, \ldots, \mathtt{x_{n1}}, \ldots, \mathtt{x_{nm}})$$

(d) b_{Ne} *returns* $(y_1, \ldots, y_m)$.

Using the definition given above, we can convert a block b into a block b_N containing only N-statements and strict operators. For example 4.4 the block b_N is:

Example 4.5:

```
f(x,y)=
  {p = eq? y 0;
   a1 =  op1(p,x);
   a2 =  op2(x);
   a =  N21(a1,a2);
   b = a + x
 in
   b}
```

The primitive operators op_1 and op_2 compute the following functions:
$op_1(p, x) = x$, when both p and x are defined, otherwise undefined.
$op_2(p) = 3$, if p is defined, otherwise undefined.

The block b_N models the non-strictness of b using strict primitive operators to compute values, and N-statements to choose among them. It is easily seen that a partitioning of the block b_N can be converted to a partitioning of the block b (all the variables in b are also present in b_N).

We now prove that the transformation is "correct" in the sense that all computations that can possibly happen at run time are modeled by the N-statement. Formally,

Theorem 4.12 (Correctness of N-block transformation) *Let e be an expression and* b_{Ne} *be the corresponding block modeling it. Then, for all bve, there is a choice of the N-statement in* b_{Ne} *such that,*

$$\mathcal{E}[\![e]\!]\; bve\; fenv = \mathcal{E}[\![b'_{Ne}]\!]\; bve\; fenv$$

where b'_{Ne} *is the block obtained by rewriting the N-statement with the appropriate choice.*

Proof: In the proof, we actually produce a choice i such that the values of the two expressions are identical.

Let $\mathcal{E}[\![e]\!]\, bve\, fenv = (v_1, \ldots, v_m)$. Define $pbve[\![x]\!] = (bve[\![x]\!], [x])$, and $abve[\![x]\!] = \{[x]\}$. By theorem 3.1 we have, $\mathcal{P}[\![e]\!]\, pbve\, penv = ((v_1, p_1), \ldots, (v_m, p_n))$ for some $p_1, \ldots, p_m$. By theorem 3.3 we have $(p_1, \ldots, p_m) \in \mathcal{A}[\![e]\!]\, abve\, aenv = P(e)$. Let i be the number of this path in $P(e)$. The choice we employ to make the theorem true is i. That is, we show that the variables $x_{i1}, \ldots, x_{im}$ have values $(v_1, \ldots, v_m)$. Consider the jth component v_j. By construction of b_{Ne} there is a statement $x_{ij} = \ldots$ in b_{Ne}. We separate two cases:

1. $p_j = \bot_p$. By theorem 3.1 $v_j = \bot$. By construction of b_{Ne}, the statement in b_{Ne} computing x_{ij} is of the form $\mathtt{x_{ij}} = \mathtt{Strict}(\ldots)$. As we have included u, the undefined variable as one of the arguments for the *Strict* operator, the value of the whole expression is $\bot$, i.e., $x_{ij} = \bot$.

2. $p_j \neq \bot_p$. There are two cases:

   (a) $v_j \neq \bot$. Let $p_j = [z_1, \ldots, z_m]$. By theorem 3.1 $bve[\![z_l]\!] \neq \bot$ for $l \in 1 \ldots m$. Thus, by the definition of op_{ij} and condition (B) of theorem 3.1, we have
   $\mathcal{E}[\![op_{ij}(bve[\![z_1]\!], \ldots, bve[\![z_m]\!])]\!]\, bve\, fenv$
   $= \mathcal{E}[\![e]\!]\, [z_i \mapsto bve[\![z_i]\!]]\, fenv$
   $= \mathcal{E}[\![e]\!]\, Proj(bve, p_j, pbve)\, fenv = v_j$

   (b) $v_j = \bot$. By corollary 3.1.1, one of $\{z_1, \ldots, z_m\}$ should be $\bot$. As op_{ij} is strict on its inputs, the value of x_{ij} is $\bot$.

□

From the above theorem it is clear that the block b_N formed by introducing N-statements can simulate any computation of the block b. Thus, to show that a partitioning is correct for the block b we only need to deal with b_N.

The proof of correctness of the AB algorithm given here is similar to the one given for correctness on strict blocks in the sense that we take a two staged approach. We define a new correctness criterion (using the block with N-statements) and prove that it is sufficient to ensure semantic correctness.

4.4.1 N-block based Correctness of a Partitioning

Definition 4.11 (N-block based correctness) *Let Q be a partitioning of a block b. Let*

b_N be the corresponding N-block of b, and $b_{N_1}, b_{N_2}, \ldots, b_{N_k}$ be all possible strict blocks which are rewritten versions of b_N. Then Q is a correct partitioning if Q is a correct partitioning of each $b_{N_i}, 1 \leq i \leq k$.

Here, we have formulated correctness of a partitioning of a non-strict block by observing that the potential dependencies get "resolved" at execution time into certain dependencies (i.e., we can regard the non-strict block as a block with strict operators). If we show that the partitioning is correct for all possible "resolutions" of potential dependencies in the block, we can prove that the partitioning is correct for the original non-strict block.

We now prove that the correctness criterion given above is enough to ensure that the semantic criterion (using *Ans*) is satisfied. First we need to prove the following lemmas. Informally, the first lemma states that the "final" environment of the block can be affected by partitioning in only one way: values of some variables may become $\perp$, but the other variables are left unchanged. The second lemma states that choices made in N-blocks are not affected by the partitioning transformation.

Lemma 4.13 *Let b be a block and b_Q its partitioned version. Let bve be the external environment and newbve, $newbve_Q$ be the internal environments computed for the blocks using bve. Then $\forall x \in BV(b)$, $newbve_Q[\![\underline{x}]\!] \sqsubseteq newbve[\![x]\!]$.*

Proof: The proof is by fixpoint induction. As usual, *newbve* and $newbve_Q$ are limits of the chains $newbve^0$, $newbve^1, \ldots$ and $newbve^0_Q$, $newbve^1_Q, \ldots$. For brevity, we shall not repeat the initial environments and the computation of successive environments here. Refer to the proof of theorem 3.1 for a similar set of equations. There are three "flavors" of variables in b_Q:

1. The original variables (denoted by x).
2. The "new" variables (denoted by $\underline{x}$).
3. The local variables (denoted by x^i, for partition i).

We prove the following properties for the environment sequences. For all x, i, k:

1. $newbve^k_Q[\![\underline{x}]\!] \sqsubseteq newbve^k_Q[\![x]\!]$.
2. $newbve^k_Q[\![x^i]\!] \sqsubseteq newbve^k_Q[\![\underline{x}]\!]$.
3. $newbve^k_Q[\![x]\!] \sqsubseteq newbve^k[\![x]\!]$.

Basis: The properties are obviously satisfied for $k = 0$ when all environments return $\bot$.
Induction: Assuming the properties for k. We prove the properties for $k+1$. Observe that $\underline{x}$ is computed by one of the two statements:

$$\begin{array}{rcl} (\ldots, \underline{x}, \ldots) & = & S_n(\ldots, x, \ldots) \\ (\ldots, \underline{x}, \ldots) & = & (\ldots, x, \ldots) \end{array}$$

in b_Q. In both cases (as they do not change the value of x), we have, $newbve_Q^{k+1}\llbracket \underline{x} \rrbracket \sqsubseteq newbve_Q^{k}\llbracket x \rrbracket$. But, as $newbve_Q^{k} \sqsubseteq newbve_Q^{k+1}$ (by monotonicity of the chains), the first property is satisfied. The proof for the second property is similar.

Consider a statement $x = e_i$ of b_Q belonging to a partition corresponding to op_i. By definition of partition, there is a statement $x = e$ in b such that $e_i = [x^i/x]e$. Thus,
$newbve_Q^{k+1}\llbracket x \rrbracket$
$= \mathcal{E}\llbracket e_i \rrbracket\, [x^i \mapsto newbve_Q^{k}\llbracket x^i \rrbracket]\, fenv$
$\sqsubseteq \mathcal{E}\llbracket e_i \rrbracket\, [x^i \mapsto newbve_Q^{k}\llbracket \underline{x} \rrbracket]\, fenv$
$\sqsubseteq \mathcal{E}\llbracket e_i \rrbracket\, [x^i \mapsto newbve_Q^{k}\llbracket x \rrbracket]\, fenv$
$\sqsubseteq \mathcal{E}\llbracket e_i \rrbracket\, [x^i \mapsto newbve^{k}\llbracket x \rrbracket]\, fenv$
$= \mathcal{E}\llbracket e \rrbracket\, [x \mapsto newbve^{k}\llbracket x \rrbracket]\, fenv$
$= newbve_Q^{k+1}\llbracket x \rrbracket$
By induction, all the properties are true for the limits of the environment chains *newbve* and $newbve_Q$. The theorem follows from the first and third property. □

Lemma 4.14 *Given a block b and an environment bve, let NC be the set of choices made at the N-statements of b_N (the N-block corresponding to b) in the environment bve. Then, given a partitioning Q and partitioned block b_Q,*

$$\mathcal{E}\llbracket b_Q \rrbracket\, bve\, fenv = \mathcal{E}\llbracket b'_{NQ} \rrbracket\, bve\, fenv$$

where b'_{NQ} is the block formed by rewriting b_{NQ} (the partitioned version of b_N) using the same set of choices NC.

Proof: Let bve_1 be the internal environment of block b computed during the evaluation and bve_2 be the internal environment of block b_Q. By lemma 4.13 we have $\forall x \in BV(b)$, $bve_2\llbracket \underline{x} \rrbracket \sqsubseteq bve_1\llbracket x \rrbracket$.

Consider an expression e which is transformed into a block b_{Ne}. Let $e_Q = [\underline{x}/x]e$ be the

corresponding expression in b_Q and b_{NeQ} the block into which e_Q is transformed.

By the property of the choices, we know that: $\mathcal{E}[\![e]\!]\, bve_1\, fenv = \mathcal{E}[\![b'_{Ne}]\!]\, bve_1\, fenv = (v_1, \ldots, v_n)$. Let $pbve$ be a value-path environment such that $pbve_1[\![x]\!] = (bve_1[\![x]\!], [x])$. We have to show that $(u_1, \ldots, u_n) = \mathcal{E}[\![[\underline{x}/x]e]\!]\, bve_2\, fenv = \mathcal{E}[\![[\underline{x}/x]b'_{NeQ}]\!]\, bve_2\, fenv$.

Let the ith choice be made at the N-statement of b_{Ne}, i.e., v_j is the value computed by the statement $x_{ij} = e_{ij}$.

Consider the jth component u_j. We show that the value of x_{ij} (in the block b_{NeQ}) is the same as u_j. By monotonicity ($bve_2 \sqsubseteq bve_1$), $u_j \sqsubseteq v_j$. We consider the following cases:

1. $p_j = \perp$. Thus, by theorem 3.1 $v_j = \perp$. By monotonicity $u_j = \perp$. Then,
   $\mathcal{E}[\![[\underline{x}/x]e_{ij}]\!]\, bve_2\, fenv$
   $\sqsubseteq \mathcal{E}[\![e_{ij}]\!]\, bve_1\, fenv$
   $= v_j = u_j$.

2. $p_j \neq \perp$. Thus, there is a statement of the form $x_{ij} = op_{ij}(z_1, \ldots, z_m)$ for $z_l \in p_j$ in b_{Ne} and a statement $x_{ij} = op_{ij}(\underline{z}_1, \ldots, \underline{z}_m)$ in b_{NeQ}. There are two cases:

   (a) $v_j \neq u_j = \perp$. By corollary 3.1.1 there is a variable $\underline{z}_l$ such that $bve_2[\![\underline{z}_l]\!] = \perp$. As op_{ij} is strict and has $\underline{z}_l$ as one of its arguments, the value of $x_{ij} = \perp$.

   (b) $v_j = u_j \neq \perp$. By corollary 3.1.1 for all $l \in 1 \ldots m$, $bve_2[\![\underline{z}_l]\!] \neq \perp$. As $bve_2 \sqsubseteq bve_1$, $bve_2[\![\underline{z}_l]\!] = bve_1[\![z_l]\!]$. Thus,
   $op_{ij}(\underline{z}_1, \ldots, \underline{z}_m)$
   $= \mathcal{E}[\![[\underline{x}/x]e]\!]\, [\underline{z}_l \mapsto bve_2[\![\underline{z}_l]\!]]\, fenv$, by definition
   $= \mathcal{E}[\![e]\!]\, [z_l \mapsto bve_2[\![\underline{z}_l]\!]]\, fenv$, by renaming
   $= \mathcal{E}[\![e]\!]\, [z_l \mapsto bve_1[\![z_l]\!]]\, fenv$,
   $= v_j = u_j$.

□

The theorems proved in this so far used denotational semantics. However, as the semantic criterion is formulated in terms of an operational semantics, we have to "switch" to operational semantics. We are justified in making this "switch" provided the following two statements are correct.

1. The two semantics are *equivalent* in some sense (that is, they produce the same answers).

2. Contexts in the operational semantics are *equivalent* to environments in denotational semantics. We can study the behavior of a block in a context by studying its behavior of the block in an equivalent environment.

Dealing with the two statements and formalizing them is beyond the scope of this thesis.

Theorem 4.15 (Sufficiency of the N-block criterion) *A partitioning Q of a block b is correct with respect to the semantic criterion if it is correct with respect to the N-block criterion.*

Proof: Let b_Q be the partitioned version of b. Let $C[]$ be any context for b. We show that $Ans(C[b]) = Ans(C[b_Q])$. By theorem 4.12 there a set of choices (NC) such that $Ans(C[b]) = Ans(C[b'_N])$. By lemma 4.14 for the same set of choices $Ans(C[b_Q]) = Ans(C[b'_{NQ}])$, where b'_{NQ} is the partitioned version of b'_N using the partition Q. As b'_N is a strict block obtained by rewriting b_N, we have (by the N-block criterion) $Ans(C[b'_N]) = Ans(C[b'_{NQ}])$, and the result follows. □

4.4.2 Correctness proof of the AB algorithm

Theorem 4.16 (Correctness of AB on non-strict blocks) *Algorithm AB correctly partitions a block with non-strict operators.*

Proof: Let b be a block and b_N be the corresponding N-block. Let BN be the set of all strict blocks obtained by rewriting b_N, the N-block corresponding to b. Let $A^s(x)$ denote the A set of a variable x produced by the block $b_s \in BN$ ($B^s(x)$ is defined similarly). The following properties are used in the proof. For all variables x, y,

$$A(x) = A(y) \Longrightarrow \forall s, A^s(x) = A^s(y)$$

$$B(x) = B(y) \Longrightarrow \forall s, B^s(x) = B^s(y)$$

We prove the two properties by induction on the number of N statements in the block. Basis: For a block with no N statements, i.e. a strict block, there is only one strict block which can be obtained by rewriting b_N, i.e. b_N itself. Thus, the A sets are same for b and for the strict block (the B sets are identical too). The theorem is trivially satisfied. Induction: Assume that the lemma is true for all blocks with less than n *N-statements*. Let b be a block with n such statements and b_N be the corresponding N-block. Let BN

be the set of strict blocks obtained by rewriting b_N. Let a statement (labeled n) be one of the N-statements such that it is the "last" in the block. That is, there is no other N-statement which uses the values produced by n. As b_N is acyclic such a statement always exists. Let the statement n have k "choices". Then, BN can be partitioned into k sets of blocks $BN^1, \ldots, BN^k$ such that the blocks in $BN^r, 1 \leq r \leq k$ correspond to the rth "choice" being made at the rewriting of the N-statement.
Let $b^1, \ldots, b^k$ be the blocks obtained by *one* rewriting of b_N at the specified N-statement. Let A^r and B^r stand for the A and B sets of variables of block b^r. Applying the induction hypothesis to the blocks $b^1, \ldots, b^k$ we get (for $r \in 1 \ldots k$), $\forall x, y$:

$$A(x) = A(y) \Longrightarrow \forall b_{r_s} \in BN^r, A^{r_s}(x) = A^{r_s}(y)$$

$$B(x) = B(y) \Longrightarrow \forall b_{r_s} \in BN^r, B^{r_s}(x) = B^{r_s}(y)$$

Thus, we only have to prove:

$$A(x) = A(y) \Longrightarrow \forall r, A^r(x) = A^r(y)$$

$$B(x) = B(y) \Longrightarrow \forall r, B^r(x) = B^r(y)$$

To prove this, we make use of the following property ($Prop(r)$):

$$\forall j, ((j, n, l) \in D \wedge (n, r) \in l) \Longleftrightarrow ((j, n-1, l \setminus \{(n, r)\}) \in D^r)$$

where the set D can stand for $A(x)$ for some variable x and D^r can stand for $A^r(x)$ (or the corresponding B sets).
The property is invariant under set union and intersection. That is, if D_1, D_1^r and D_2, D_2^r satisfy the property for some r, then so do $(D_1 \cup D_2), (D_1^r \cup D_2^r)$ and $(D_1 \cap D_2), (D_1^r \cap D_2^r)$.
We now prove the property is true for A sets. That is,

$$\forall j, ((j, n, l) \in A(x) \wedge (n, r) \in l) \Longleftrightarrow ((j, n-1, l \setminus \{(n, r)\}) \in A^r(x))$$

Note that a label denotes the set of "choices" made at the N-statements to "reach" a variable from the output. Let α denote (j, n, l) and β denote $(j, n-1, l' \ \{(n, r)\})$. We consider two cases (refer to figure 4.3):

1. α reaches $A(x)$ without using any of the edges corresponding to the non-strict operator n. Then, this path will be present in b^r too, proving $\beta \in A^r(x)$. The proof of the converse is similar.

2. Assume α "uses" one of the k choices to reach x. The only choice it can use is the rth choice, as $(n,r) \in l$. As this choice is "wired" in b^r, $\beta \in A^r(x)$. Again, the proof of the converse is similar.

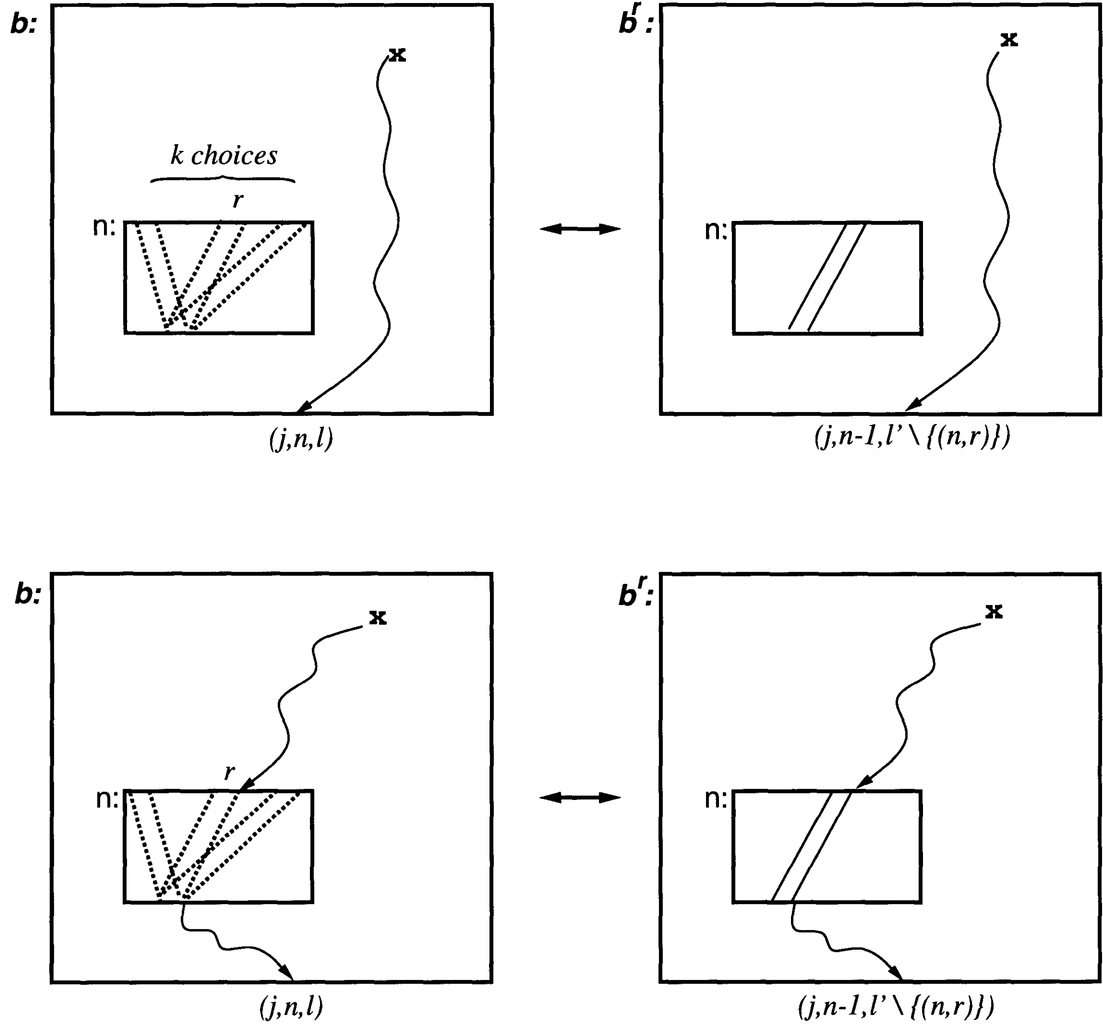


Figure 4.3: Proof of $Prop(r)$ for A sets

The proof of $Prop(r)$ for B sets is by induction on the structure of the dependence graph.

Basis: For input variables, the B sets are the same as A sets and thus $Prop(r)$ is satisfied.

Induction: Assume we are computing B sets at some statement. There are two cases.

1. We are at a primitive operator using an expression e. Here, $B(y) = \bigcap_{x \in FV(e)} B(x)$ and $B^r(y) = \bigcap_{x \in FV(e)} B^r(x)$. By induction hypothesis, $Prop(r)$ is true for each x. The result follows by invariance over intersection.

2. We are at a non-strict operator. There are two cases:

   (a) We are not at the statement labeled n. Let the statement be labeled n'. For a variable y, the B set is of the form $B(y) = \bigcup_r (\cap_x B(x) \cap L_{(n',r')})$. The computation of $B^r(y)$ is identical; only $B(x)$ is replaced by $B^r(y)$ etc. It is easy to verify that $Prop(r)$ is true for the L sets too. Thus, the fact that $B(y), B^r(y)$ satisfy the property follows from the invariance over $\cup$ and $\cap$ operators.

   (b) We are at the statement labeled n. Then the computation of $B(y)$ and $B^r(y)$ sets can be written as follows:

$$B(y) = (T_1 \cap L_{(n,1)}) \cup \cdots \cup (T_k \cap L_{(n,k)})$$
$$B^r(y) = T_r^r$$

   The T's in the above equations stand for intersection of B sets of some variables (and hence satisfy $Prop(r)$, by induction hypothesis and invariance over intersection). The equation for $B^r(x)$ reflects the fact that the rth choice is "hard-wired" in the computation of $B^r(x)$. We show that $Prop(r)$ is valid for $B(x)$.

$$\forall j, ((j,n,l) \in B(x) \wedge (n,r) \in l) \Longleftrightarrow ((j, n-1, l \setminus \{(n,r)\}) \in B^r(x))$$

      i. $(\Rightarrow)$: $(j,n,l) \in (T_i \cap L_{(n,i)})$, for some i: The only possibility is $i = r$ as only $L_{(n,r)}$ contains such labels. Thus, $(j,n,l) \in (T_r \cap L_{(n,r)}) \Longrightarrow (j,n,l) \in T_r$ $\Longrightarrow (j, n-1, l \setminus \{(n,r)\}) \in T_r^r$. Thus, $(j, n-1, l \setminus \{(n,r)\}) \in B^k(x)$.

      ii. $(\Leftarrow)$: $(j, n-1, l') \in T_r^r$ implies $(j, n-1, l' \cup \{(n,r)\}) \in T_r$. Thus, $(j, n-1, l' \cup \{(n,r)\}) \in (T_r \cap L_{(n,r)})$. Thus, $(j, n-1, l' \cup \{(n,r)\}) \in B(y)$.

Using $Prop(r)$ for A sets, we prove that $\exists r, (A^r(x) \neq A^r(y)) \Longrightarrow (A(x) \neq A(y))$. Assume $\exists r, (A^r(x) \neq A^r(y))$. Then, without loss of generality,

$$\exists (j, n-1, l') \in A^r(x) \text{ such that } (j, n-1, l') \notin A^r(y)$$

Thus, $(j, n-1, l' \cup \{(n,r)\}) \in A(x)$ and $(j, n-1, l' \cup \{(n,r)\}) \notin A(y)$. Thus $A(x) \neq A(y)$. The proof for B sets is similar.

So far we have proved the following properties relating the A and B sets computed for b to those computed for a strict block.

$$A(x) = A(y) \Longrightarrow \forall s, A^s(x) = A^s(y)$$

$$B(x) = B(y) \Longrightarrow \forall s, B^s(x) = B^s(y)$$

Now, we show that the partitioned block b_Q is acyclic. This implies that each partitioned strict block is acyclic, as they contain a subset of the edges of the dependence graph. Thus, the correctness criterion for strict blocks is satisfied, and by the N-block criterion, the correctness for non-strict block b is established.

We show (proof by contradiction) that subpartitioning ensures that the partitioned block b_Q is acyclic. Assume that we have formed partitions using the A stage of the AB algorithm. Let $q_1, \ldots, q_n$ form a cycle in b_Q. There are two cases:

1. The paths in the dependence graph of b between every two successive partitions in the cycle are of weight 0. Thus, the paths are all strict. By theorem 4.11 we have, $A(q_1) \supseteq A(q_2) \cdots \supseteq A(q_n) \supseteq A(q_1)$. Thus, the A sets are all equal and they would have been put in the same partition.

2. One of the paths is of weight ≥ 1. Let $x, y \in q_1$ such that there is a path (in the dependence graph of b) from x to some variable in q_2 and there is a path from some variable in q_n to y. Thus, if z is the variable in q_2 which is connected to x, $depth(z) \geq depth(x)$ as the depths increase monotonically along any path. As the depths of all variables in q_i are equal for each i, we can associate that depth with q_i. Thus $depth(x) \leq depth(q_2) \leq \ldots \leq depth(q_n) \leq y$. At least one of the inequalities must be strictly $<$, as there is a path of weight ≥ 1. Thus $depth(x) < depth(y)$, a contradiction.

□

4.4.3 Summary

In this section, we have extended the AB algorithm to work on non-strict blocks. The key idea is to observe that eventually (i.e., at run time) the non-strict block "resolves" itself to a strict block. The set of all possible strict block it can resolve to can be determined by paths of the non-strict expressions in the block. The AB algorithm conceptually runs the strict block version on each of the strict blocks generated.

The correctness proof consists of two parts. One is that paths accurately describe the set of all strict blocks and the second is to show that we have extended the AB algorithm

correctly, i.e., its behavior is consistent with the algorithm running on each individual strict block.

4.5 Appendix: Complexity of Partitioning

Theorem 4.17 (Partitioning is hard) *Optimal partitioning is NP-hard.*

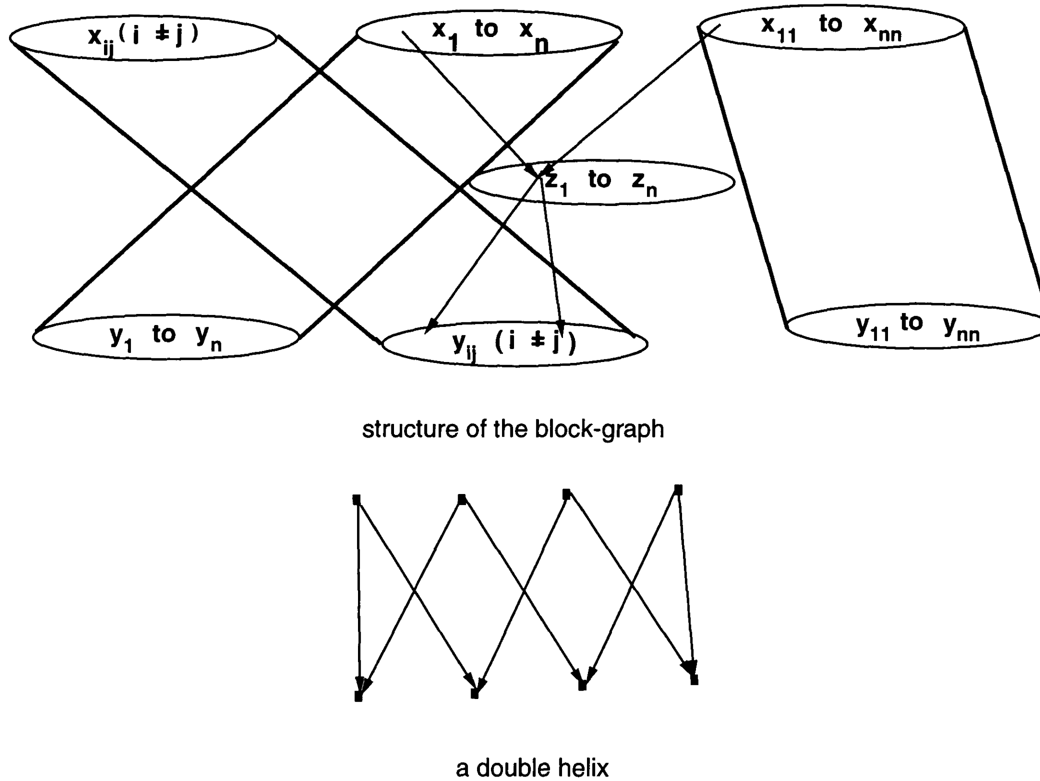


Figure 4.4: Proof of NP-hardness

Proof: We reduce the NP-hard problem of finding optimal (minimum number of) clique partitions of an undirected graph [14] to partitioning. Given an undirected graph $G = (V, E)$ with $n \geq 4$ vertices, we construct a dependence graph (which can be easily converted into a block with strict operators) as follows:

$G_{db} = (V_b, E_b)$ where,

$V_b = \{x_1, \ldots, x_n\} \bigcup$

$\{x_{11}, \ldots, x_{1n}, \ldots, x_{n1}, \ldots, x_{nn}\} \bigcup$

$\{y_1, \ldots, y_n\} \bigcup$

$$\{y_{11},\ldots,y_{1n},\ldots,y_{n1},\ldots,y_{nn}\}\bigcup$$
$$\{z_1,\ldots,z_n\}$$
$$E_b = \{(z_i, y_{ij}), 1 \le i,j \le n, j \ne i\}\bigcup\{(x_i, z_i), (x_{ii}, z_i), 1 \le i \le n\}\bigcup$$
$$\{(x_i, y_{kj}), (x_{ii}, y_{kj}), 1 \le i,j,k \le n, \text{ such that}$$
$$(k \ne j, j = i, (k,j) \in E)\bigvee(j \ne i, k \ne j))\}$$

We also have additional edges forming "double-helix" between each of the following pairs of vertex sets $(\{x_1,\ldots,x_n\},\{y_1,\ldots,y_n\})$, $(\{x_{11},\ldots,x_{nn}\},\{y_{11},\ldots,y_{nn}\})$, $(\{x_{ij}|1 \le i,j \le n, i \ne j\},\{y_{ij}|1 \le i,j \le n, i \ne j\})$. The property of a "double-helix" is that it forces each variable in it to be in different partitions. We make the following observations:

1. x_i, x_{jj} cannot be put in the same partition. This follows from the fact that $\exists y_k \in A(x_i)$ and $y_k \notin A(x_{jj})$. Thus, $y_k \notin B(x_{jj})$, as $B(x_{jj}) = A(x_{jj})$. Similarly x_i and x_{jk}, x_{ii} and x_{jk} cannot be put in the same partition.
2. For any output y, $A(y) = \{y\}$ and $B(y) = \{y\}$. This follows from the double-helix construction, forcing each output to be in a separate partition. Also, any other variable w cannot be put in the same partition as any output y, as $|A(w)| > 1$ and $|B(y)| = 1$.
3. x_i, z_j cannot be put in the same partition, as $\exists y_k \in A(x_i)$ and $y_k \notin B(z_j)$ (follows from the fact that y_k is connected only to x_i's not x_{ii}'s). Similarly x_{ii}, z_j cannot be put in the same partition. z_i, x_{jk} cannot be put in the same partition as $|A(z_i)| \ge 3$ and $|A(x_{jk})| = 2 = |B(x_{jk})|$.

From the observations mentioned above, it is clear that the only non trivial (having more than one variable) partitions can be formed by putting sets of z_i's together in the same partition. Also, note that including some z_i's in the same partitions does not change the A and B sets of other z_i's (follows from the fact that there is no path from any z_i to z_j). Thus, any clique partitioning of the graph $G_z = (V_z, E_z)$, where $V_z = \{z_1,\ldots,z_n\}$ and $E_z = \{(z_i,z_j)|(A(z_i) \subseteq B(z_j))\bigwedge(A(z_j) \subseteq B(z_i))\}$ generates a partitioning of the dependence graph constructed and vice-versa. We now show that the graph G_z is isomorphic to G, with the isomorphism $i \leftrightarrow z_i$:
$A(z_i) = \{y_{i1},\ldots,y_{in}\}$. But, $y_{ik} \in B(z_j)$ for $k \ne j$ (as $y_{ik} \in x_i$ and $y_{ik} \in x_{ii}$).
Thus, $A(z_i) \subseteq B(z_j)$ iff $y_{ij} \in B(z_j)$. A similar statement is true for z_j.

Thus, $(z_i, z_j) \in E_z$ iff $(y_{ji} \in B(z_i)) \bigwedge (y_{ij} \in B(z_j))$ iff $(i, j) \in E$.

This completes the reduction, and the NP-hardness follows. □

Note: In the construction given, we have used primitive operators having an arbitrary number of inputs. If we restrict all primitive operators to have two inputs, then we can simulate the construction by using a "tree" of such two input operators.

Chapter 5

Extensions

In the previous chapter we have given a partitioning algorithm to partition first order, functional programs with simple data types. The algorithm can be improved in several ways:

1. The algorithm partitions for a worst case scenario; that is, it assumes that a block can be called in an arbitrary context. We can probably produce better partitions by using the information present at the places where it is called.

2. The algorithm does not handle data-structures. That is, dependence information is not propagated through data-structures. Also, it does not partition programs with side-effects in it.

3. The algorithm does not handle higher order functions.

In this chapter, we attempt to provide (partial) solutions to the first two problems.

5.1 Partitioning with Contexts

In this section we will try to partition a program in its entirety, so that we may be able to produce better partitions. The intuition behind this section is simple: a function or a block is rarely used in all possible contexts, it is probably only used in a small fraction of them. Thus, by considering only those contexts in which a block is used, we might partition better. If no context in which the function is called makes use of the non-strict behavior of the function, we should be able to compile the function in a strict manner. In some sense, this is also "fair" to the writer of the program; there is a performance penalty only

if he uses the non-strictness of a function. However, we point out that this is the "ideal" case which we want to achieve, but we might not be able to realize this due to the various approximations we have to make at compile time.

The following example illustrates our algorithm to make use of context information.

Example 5.1:

```
f(x,y)=
  { a = x + 1;
    b = y * 2
  in
    (a,b)}
g(z)=
  { (r,s) = f(z,z)
  in
    (r,s)}
```

Suppose we know that the function f is called only in the function g, we can conclude that the operators + and * can be put in the same partition, as the inputs of both operators come from the same "value" (i.e., z of function g). This argument can be formalized in the partitioning algorithm: as we know the path information about function f, and the inputs to it in function g, we obtain a partition where r and s are in the same partition. That is, there is a statement $(r1, s1) = S(r, s)$ in the partitioned version of g. Moving this statement "inward" (i.e., into f), we introduce a statement $(a1, b1) = S(a, b)$ in f (which now returns $(a1, b1)$). The partitioning algorithm working on the new definition of f will succeed in obtaining the desired partitions. The rest of the section gives the algorithm to propagate context information.

Definition 5.1 (P-Contexts) *Let b be a block with input variables $X = \{x_1, \ldots, x_n\}$ and output variables $Y = \{y_1, \ldots, y_n\}$. A* partitioning context *(or simply, p-context) is a pair of sets (PC_s, PC_r) (the "send" and "receive" sets), such that PC_1 partitions the set X and PC_2 partitions the set Y.*

Informally, each partition in a p-context denotes the inputs (or outputs) which can be put in the same partition. For the example given above, the p-context of f is $(\{\{x\}, \{y\}\}, \{\{a, b\}\})$.

We now prove that propagating context information "inside" is "safe".

Theorem 5.1 (Correctness of context propagation) *Let b_1 be a block used* only *in another block b_2. Let there be statement of the form $(\mathtt{x_1}, \ldots, \mathtt{x_n}) = \mathtt{S}(\mathtt{x'_1}, \ldots, \mathtt{x'_n})$ in b_2,* where $\{x_1, \ldots, x_n\} \subseteq FV(b_1)$. Then, the transformation of the block b_1 into b'_1, where

1. There is a statement $(\mathtt{y_1}, \ldots, \mathtt{y_n}) = \mathtt{S}(\mathtt{x_1}, \ldots, \mathtt{x_n})$ added to b_1.

2. Each occurrence of x_i in b_1 is replaced by y_i, for $1 \leq i \leq n$.

is correct.

Proof: The notion of correctness used in this proof is similar to the one used for partitioning. Let b_2' be the block which uses b_1' instead of b. We have to prove that $\forall C, Ans(C[b_2]) = Ans(C[b_2'])$, where C is any user defined context. Due to the property of adding the S statement (which is similar to the partitioning transformation), we need to prove that if $C[b_2]$ terminates properly, so does $C[b_2']$. There are two cases:

1. If C is such that the additional S statement in b_1 gets rewritten, then the blocks b_2 and b_2' are identical. Thus, they yield identical answers in such a context C.

2. Assume that the S statement in b_1' is not rewritten. Thus, $C[b_2']$ is a deadlock. By the behavior of the S statement, one of $\{x_1, \ldots, x_n\}$ must undefined. As they all depend on the same S statement (in b_2) for a value, the S statement in b_2 must not have been rewritten. Thus, $C[b_2]$ is also a deadlock.

□

Note: A similar theorem can be proved about the "outputs" of the block b_1.

Using the theorem, we can determine a p-context for each block based on the context information (for blocks that are used only once). However, in a program, it is very unusual that a block (or a function) is used only in one context; in general, it will be used in multiple contexts. If we need to partition the block so that it is safe to call the block in all the contexts, we need to be able to find a safe approximation to all the p-contexts. The following definition gives a method of comparing information in contexts and computing approximations of them. Basically, a p-context approximates another p-context if the partitions in the p-context "refines" the partitions of the other p-context.

Definition 5.2 *A p-context $PC_2 = (PC_{2s}, PC_{2r})$ approximates another p-context $PC_1 = (PC_{1s}, PC_{1r})$ (i.e. $PC_2 \preceq PC_1$) iff*
$\forall pc_{2s} \in PC_{2s}, \exists pc_{1s} \in PC_{1s}$ such that $pc_{2s} \subseteq pc_{1s}$ and
$\forall pc_{2r} \in PC_{2r}, \exists pc_{1r} \in PC_{1r}$ such that $pc_{2r} \subseteq pc_{1r}$.

Using the definition above, we can compute the greatest lower bound (*glb*, denoted by the symbol $\prod$) as follows:

If $PC_1 = (PC_{1s}, PC_{1r})$ and $PC_2 = (PC_{2s}, PC_{2r})$ are two p-contexts
then $PC_3 = (PC_{3s}, PC_{3r}) = PC_1 \prod PC_2$ where
$pc_{3s} \in PC_{3s}$ iff $(pc_{3s} \neq \phi) \wedge (\exists pc_{1s} \in PC_{1s}, pc_{2s} \in PC_{2s},\ pc_{3s} = (pc_{1s} \cap pc_{2s}))$.
PC_{3r} is computed in a similar manner.
It is straightforward to show that PC_3 is indeed the greatest lower bound.

Theorem 5.2 (Approximating multiple contexts) *Let $b_1, \ldots, b_n$ be blocks using a block b. Let $PC_1, PC_2, \ldots, PC_n$ be the p-contexts of b in $b_1, \ldots, b_n$. Then transforming the block (introducing S statements) according to the p-context $PC = \prod_{1 \leq i \leq n} PC_i$ is correct.*

Proof: Observe that the proof of theorem 5.1 holds even if we use an approximation to the p-context defined in the outer block (i.e., instead of a single S statement we introduce several "smaller" ones). Thus, the p-context PC is safe with respect to all the blocks $b_1, \ldots, b_n$, as it is a lower bound on the p-contexts $PC_1, \ldots, PC_n$. The correctness of the transformation follows easily from this property. □

So far, we have not dealt with the case when the context information we are propagating affects the context itself. This can happen in the case of recursive functions. In such cases, the context information to be propagated can be obtained by fix point iteration. We give an example to outline our approach.

Example 5.2:
```
f(x,y) =
{p = eq? y 0;
 r = if p  then
               { in 1}
           else
               {t1 = y - 1;
                t2 = x;
                t3 = f(t2,t1);
               in
                 t3}
 in
   r}
```

Note that the two arguments to f cannot be included in the same partition (in general), as a context such as $\mathtt{a} = \mathtt{f(a,0)}$ will deadlock. However, for this example, assume that we can conclude from the contexts calling f that the two arguments can be put in the same partition. Thus, we can introduce a statement $(\mathtt{x_1}, \mathtt{y_1}) = \mathtt{S(x,y)}$ in f. Using this, we can

conclude that the "send" p-context of the recursive call to f is $\{\{t1, t2\}\}$. The *glb* of the two p-contexts (the outer and the recursive) yields the same p-context as the result. That is, we have reached a fixed point, and the partitioning of f using this p-context is correct. We now specify the algorithm to do fix point iteration.

Algorithm R

Given a recursive function f:

1. Let pc be the *glb* of the p-contexts calling f.

2. Do until the p-context pc does not change.

   (a) Partition the body of f (using algorithm C and algorithm AB).

   (b) Let $pc_1, pc_2, \ldots, pc_n$ be the p-contexts of the recursive calls to f. Let

$$pc = pc \prod (\prod_{1 \leq i \leq n} pc_i)$$

Note: The algorithm generates a *decreasing* chain of p-contexts. As the longest chain can at most be of length $m + n$ (m = no. of arguments, n = no. of results), the algorithm terminates after at most $m + n$ steps.

Theorem 5.3 *Algorithm R correctly propagates context information to a recursive function.*

Proof: Let $pc_0, pc_1, \ldots, pc_n, \ldots$. be the p-contexts generated by algorithm R. We prove that, after i stages, the context information propagated is correct for all calls to f that are at most i deep in the recursion.

Basis: If there is no recursive call to f, pc_0 is the correct context (according to theorem 5.2).

Induction: Assume that the hypothesis is true for i. Consider a recursive call to f which $i + 1$ deep in the recursion. The contexts in which this call occurs are obtained by:

1. Using the p-context (pc_i) at the ith level (which is correct, by induction hypothesis).

2. Partitioning the body of f (which is correct as we are using algorithms C and AB).

Thus, pc_{i+1} is correct for any call which is $\leq i$ deep, as $pc_{i+1} \preceq pc_i$. It is also correct for a recursive call at depth $i + 1$ as any such context $pc \succeq pc_{i+1}$. This proves the hypothesis, and the theorem follows. □

Note: We mention here that mutual recursion can be handled in a similar manner; we use simultaneous fixed point iteration to generate the p-contexts.

So, the strategy for taking into account context-information is this: Given the call graph of a program, we descend from the root to the leaves, resolving the contexts in a cycle in the call graph using fix point iteration.

Finally, we point out a drawback of our representation as of context information as p-contexts. Though we get the benefit of fast convergence in fixed point iteration, we are losing correlation information between input (or output) variables. That is, even though two input variables may not be put in the same partition in a context, there might be enough correlation in the values of the variables to enable bigger partitions of the block. One solution would be to propagate the A and B sets themselves into a block (from the surrounding block). However, the method to handle recursion in that case is not very obvious. This is an area of future research.

5.2 Data Structures

In this section, we give a brief overview of extensions to the basic algorithm which can be used to model data-structures. Our extensions have a limitation that they cannot handle data structures lists, trees etc., which could have an arbitrary number of components associated with them. The following example shows one program which we can analyze using the algorithm.

Example 5.3:

```
f(x,y) =
  {a = x + 1;
   b = y - 1;
   c = mk_tuple2(a,b)
 in
   c}
g(a,b,c) =
 {t = f(a,a);
  t1 = select1(t);
  t2 = select2(t);
 in
   (t1,t2)}
```

The **mk_tuple**$_n$ primitive operator takes n values as its arguments and returns an n-tuple containing those values. There is another operator **select**$_i$ which selects the ith component of a tuple. In the example shown, we should be able to deduce that $t1$ and $t2$

can be computed in the same thread. This can be achieved by deducing that $t1$ and $t2$ depend on the same value z. Thus, we should "track" dependencies through the **mk_tuple** and **select** operators.

First, we extend the abstract path semantics to give useful information about tuples. Then, we provide methods of using this information in the partitioning algorithm.

5.2.1 Abstract Paths for Data-structures

Recall that a path of an expression is the set of free variables of the expression needed to produce a value for the expression. A simple extension to expressions which return data-structures is to include in the path of the expression, the path of each of its "components". Obviously, this leads to problems when dealing with data-structures like lists, where the value of an expression can have an unbounded number of components. Thus, we have to restrict the analysis to work on programs having tuples of some maximum "depth". We treat data structures like lists and trees by desugaring them using side-effects and using the results of the next section to make conservative approximations.

The idea is the following: a value is now a tuple – one standing for the top level value, and the other standing for the "components" of the value. A path (p') in this framework is of the form $p' = (p, \langle(1, p'_1), \ldots, (n, p'_n)\rangle)$, where p is a path as defined in chapter 3. Here p corresponds to the path of the top level value, and p'_i corresponds to the path of the ith component of the value. For simple data types like integers, p' will be of the form $(p, \langle\rangle)$ reflecting the fact that such values do not have any components associated with them. The new abstraction equations which are different from the path equations for simple data types are given below:

$$\begin{aligned}
\mathcal{A}_k[\![+]\!] &= \lambda s.\{(x:y, \langle\rangle) | ((x, xs), (y, ys)) \in s\} \\
\mathcal{A}_k[\![\mathbf{mk_tuple_n}]\!] &= \lambda s.\{([], \langle(1, x'_1), \ldots, (n, x'_n)\rangle) | (x'_1, \ldots, x'_n) \in s\} \\
\mathcal{A}_k[\![\mathbf{select_i}]\!] &= \lambda s.\{((i, (b, bs)) \in as) \rightarrow (a:b, bs), (\perp_p, \langle\rangle) | (a, as) \in s\} \\
\mathcal{A}_s[\![c]\!]\, abve &= \{([], \langle\rangle)\}
\end{aligned}$$

$\mathcal{A}[\![\mathbf{if}\ se\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2]\!]\ abve\ aenv =$

let $(x, xs) \in \mathcal{A}_s[\![se]\!]\ abve$

$((y_1, ys_1), \ldots, (y_n, ys_n)) \in \mathcal{A}_b[\![b_1]\!]\ abve\ aenv$

$((z_1, zs_1), \ldots, (z_n, zs_n)) \in \mathcal{A}_b[\![b_2]\!]\ abve\ aenv$

in $\{((x : y_1, ys_1), \ldots, (x : y_n, ys_n)), ((x : z_1, zs_1), \ldots, (x : z_n, zs_n))\}$.

Discussion:

1. The equation for + is similar to the equation for + in the path semantics without tuples. Along with the path for the value, it returns the "empty" component set. The rule for constants is similar.

2. The $\mathbf{mk_tuple_n}$ operator returns an empty path as a top level value reflecting the non-strict behavior of the tuple. None of its components need to be defined for the tuple to be defined. It also returns a component set containing the various components of the tuple.

3. The equation for **select** returns the ith component if it is defined (i.e., there is an element of the form (i, b') in the component set). Otherwise, it returns $\bot_p$. The top level path also includes the top level path of the argument; a tuple needs to be defined for selecting a component out of it.

4. The rule for **if** chooses the appropriate components (depending on the value of the predicate), along with the top level values of the result.

5.2.2 Partitioning with data-structures

First, we need to produce paths for an expression of a program which capture the behavior of the expressions. This is similar to the method used for simple data types. The path of an expression e with respect to a program pr ($P(e)$) is $\mathcal{A}[\![e]\!]\,[x \mapsto \{\overline{x}\}]\,\mathcal{A}_p[\![pr]\!]$. We use $\overline{x}$ to denote the path $([x], \langle(1, \overline{x.1}), \ldots, (n, \overline{x.n})\rangle)$, for some n where $\overline{x.i}$ is defined similarly. As our data structures have finite width (n is finite at each "level") and finite depth, this representation is finite. Using this notation the paths for a few expressions are given below:

$$
\begin{aligned}
P(\mathbf{mk_tuple_2}(x, y)) &= \{([], \langle(1, \overline{x}), (2, \overline{y})\rangle)\} \\
P(+(x, y)) &= \{([x, y], \langle\rangle)\} \\
P(\mathbf{select_1}(a)) &= \{([a, a.1], \langle(1, \overline{a.1.1}), \ldots, (n, \overline{a.1.n})\rangle)\}
\end{aligned}
$$

Note that the paths for expressions that do not use the data-structure operations will be identical to the paths defined in earlier chapters.

Given a block b we transform it to another block b' containing only top level paths, making sure that all variables present in b are present in b'. Then, we can apply the AB algorithm to partition block b', and convert these partitions to a partitioning of b. The intuition behind this transformation is that when we put a variable in a partition, we "wait" for only the top level value of that variable. Thus, we can partition using only top level paths of variables. One thing the transformation should do is to identify all variables which are being returned as part of a data structure. In the transformed block b' these will be *explicitly* returned by b'. The transformation algorithm is defined below:

Algorithm D

Given a block b:

1. Transform b to a block b_P such that every variable (except those variables defined by an N-statement) is associated with a single path. This is very similar to the N-block transformation (definition 4.10).

2. Each $x = p'$ (a path statement) in b_P is replaced in b' by the statements generated by the following (recursive) procedure.

$$\begin{aligned} Statements(x, \overline{y}) &= \{(x = \overline{y})\} \\ Statements(x, ([p], \langle(1, p'_1), \ldots, (n, p'_n)\rangle)) &= \{(x = p)\} \bigcup_{1 \leq i \leq n} Statements(x.i, p'_i) \end{aligned}$$

3. (Determine "reachable" variables) If b_N returns $(t_1, \ldots, t_m)$ then $R = \{t_1, \ldots, t_n\}$. Apply the following rules to determine all reachable variables:

   (a) If $x \in R$ then $x.\alpha \in R$.

   (b) If $x.\alpha.\beta \in R$ and $x.\alpha = \overline{y}$ is a statement in b', then $y.\beta \in R$.

   (c) If there is a statement $(y_1, \ldots, y_n) = \{(x_{11}, \ldots, x_{1n}), \ldots, (x_{m1}, \ldots, x_{mn})\}$ in b' and $y_i \in R$, then for all $1 \leq j \leq m$, $x_{ji} \in R$.

   Replace variables $\overline{x}$ by $[x]$ for each $x \in b'$. b' returns $(t_1, \ldots, t_m, r_1, \ldots, r_k)$ where $R = \{r_1, \ldots, r_k\}$.

4. Partition b' using algorithms AB and C.

The various steps of algorithm D are explained below:

1. The first part replaces statements like $x = \{p_1, \ldots, p_n\}$ with n variables computing each path individually and an N-statement selecting a single value out of n values. This easily generalizes to a statement computing multiple values.

2. The second step brings all paths to the top level by creating new variables which stand for "components".

3. The third part traces through data structures and N-statements to find out which variables are actually being returned by the block (through some data structure).

Consider example 5.3. The block path statements in the block b' are (assuming that tuples have at most 2 components):

$$\begin{aligned} t &= [] \\ t.1 &= a \\ t.2 &= a \\ t1 &= [t, t.1] \\ t1.1 &= \overline{t.1.1} \\ t1.2 &= \overline{t.1.2} \\ t2 &= [t, t.2] \\ t2.1 &= \overline{t.2.1} \\ t2.2 &= \overline{t.2.2} \end{aligned}$$

Observe that algorithm AB will now put $t1$ and $t2$ in the same thread, as their dependence on a has been made *explicit* via the variables $t.1$ and $t.2$.

Consider the following example:

```
g(a,b) =
  {t1 = a + b;
   t2 = mk_tuple2(a,b);
  in
    (t1,t2)}
```

In this example, it is not safe to put a and b in the same thread as they are returned using the tuple $t2$. This shows up in algorithm D as follows. Block b' is:

$$t1 = [a + b]$$

$$
\begin{aligned}
t2 &= [] \\
t2.1 &= \overline{a} \\
t2.2 &= \overline{b}
\end{aligned}
$$

Now, as $t2$ is reachable, so are $t2.1$ and $t2.2$. This implies that a and b are also reachable. Hence, block b' returns $(t1, t2, a, b)$ ensuring that a and b are put in different threads.

5.3 Side Effects

We do not attempt to propagate dependence information through side effects. However, we give path equations which are still "safe" in the presence of side effects – though they do not provide additional information. The modifications are very similar to the way DD handles side affects by being conservative about dependence information it computes in the presence of side effects. The approximations are achieved by using $\top$. This stands for an "over-defined" variable, such that the B set of $\top$ ($B(\top)$) is empty always.

In the new set of equations, the meaning of an expression is a pair consisting of the usual set of paths and a set of "stores" – which is the paths of all variables stored anywhere in the expression. The relevant semantic equations are given below:

$$\mathcal{A}_k[\![\mathbf{I_alloc_n}]\!] = (\{([], \{(i, \top) | 1 \leq i \leq n\})\}, \{\})$$

$$
\begin{aligned}
&\mathcal{A}_b[\![\{\{(y_{i1}, \ldots, y_{im}) = e_i\}^* \; \{\mathbf{store}_k(u_j, w_j)\}^* \; \mathbf{in} \; (se_1, se_2, \ldots, se_n)\}]\!] \; abve \; aenv = \\
&\quad \text{letrec } newabve = [y_{ij} \mapsto ((\mathcal{A}[\![e_i]\!] \; (newabve.abve) \; aenv) \downarrow 1) \downarrow j] \\
&\qquad Sr_i = (\mathcal{A}[\![e_i]\!] \; (newabve.abve) \; aenv) \downarrow 2 \\
&\qquad newabve' = [y_{ij} \mapsto \{p_{ij}\}], \text{ such that } p_{ij} \in newabve[\![y_{ij}]\!] \text{ and} \\
&\qquad\qquad\qquad p_{ij} \in ((A[\![e_i]\!] \; newabve' \; aenv) \downarrow 1) \downarrow j \\
&\qquad (u_j, us_j) = newabve'[\![u_j]\!] \\
&\qquad (v_j, vs_j) = newabve'[\![v_j]\!] \\
&\qquad sr_j = (u_j : v_j, vs_j) \\
&\quad \text{in } (\mathcal{A}_s[\![se_1]\!] \; newabve' \times \cdots \times \mathcal{A}_s[\![se_n]\!] \; newabve'), (\textstyle\bigcup_i Sr_i \bigcup_j \{sr_j\})
\end{aligned}
$$

The equation for **I_alloc** returns an empty path for the top level path and a component set where every component is $\top$. This models the fact that we do not know (in general) at compile time what values could be stored in the tuple. The equation for a block "collects"

all the stores in the block as well as the stores in the subexpressions and returns these stores along with the paths for the block.

We have remarked before that we can handle data structures like lists etc. using side effects. This is easily achieved by replacing a **cons** operator (which allocates a list element) into an **I_alloc** operator and a sequence of stores. This is similar to the way a **cons** is handled in the operational semantics of P-TAC [4].

To deal with operators like **head** and **tail** which select one of the two components present in a list element, we use the following equation

$$\mathcal{A}_k[\![\mathbf{head}]\!] = \lambda s.\{(a : \top, \{(1, \top), (2, \top)\} | (a, as) \in s\}$$

As we have not retained information about the various components of a list, we conservatively return $\top$.

The partitioning algorithm needs to be modified slightly; In addition to the propagation of backward propagation of labels to compute A sets, we also do the following: each "store" in the block is associated a new label, and is propagated to each variable in the path corresponding to the store. Essentially, we are dealing with stores by considering them as additional "outputs" of the block. The labels generated by these "stores" have exactly the same structure and behavior as labels generated by outputs.

5.4 Summary

In this chapter, we have extended the basic algorithm in three directions:

1. Make use of context information during partitioning: Intuitively, we propagate information into function bodies about arguments or results which can be put together. For recursive functions, we have to iterate before we can get a safe approximation to the contexts in which the function can be called.

2. Handling simple data structures: This extension captures dependence through data structures in paths and uses them to do partitioning.

3. Handling side effects: This is just to ensure safety in presence of side effects, it *does not* generate new dependence information.

Chapter 6

Conclusion

In this thesis, we have presented a partitioning algorithm for a first order non-strict language. Our algorithm improves the previous algorithms in several directions: we can handle recursion, produce better partitions, and partially handle data structures. Our algorithm uses paths to summarize dependence information and uses it to do partitioning. One other advantage of our algorithms is that label sets do not grow with the problem size (assuming that the number of arguments to a function is bounded by a constant). This is in contrast with previous algorithms, where the label sets grew with the problem size.

Our proof technique to prove correctness of these algorithms is based on a two-stage approach. We formulate correctness criteria based on the *structure* of the program, which yields to effective and efficient algorithms. We also prove these criteria are valid with respect to the operational and denotational semantics of the language.

6.1 Further Research

There remain many interesting avenues to explore in this field.

6.1.1 Implementation

We have only presented algorithms in our thesis. The next step would be to implement the algorithms and compare them with previous partitioning algorithms. This will shed light on the relative costs of implementing various features of a non-strict language.

6.1.2 Higher Order Functions

Our analysis makes conservative approximations when encountered with a higher order functions. An analysis which handles higher order functions efficiently would greatly benefit programs which uses such functions.

6.1.3 Recursive Data Types

In strictness analysis, there have been various suggestions on how to deal with data structures like lists, trees etc. These work by approximating the information represented – or in technical terms, compressing the *abstract domain* safely. Consider the following functions defined on lists (written in Id).

```
def nil? nil = true |
    nil? _   = false;
def length nil    = 0 |
    length (x:xs) = 1 + length xs;
def sum nil    = 0 |
    sum (x:xs) = x + sum xs;
```

Strictness analysis on lists yields the following information about a function which uses lists.

1. The function is not strict in the list.
2. The function is strict in the top-level value of the list. An example of such a function is `nil?`.
3. The function is strict in *all* the tails of the list. For example, the function `length` which returns the length of a list evaluates all the "tails" of the list.
4. The function is strict in *all* the heads and tails of the list. The function `sum` which sums up all the elements present in a list exhibits this property.

It is open whether such information can be use effectively in our partitioning algorithms.

6.1.4 Subscript Analysis

Our partitioning algorithm can be extended to give safe partitions on programs using arrays by treating them as we treat side effect operations. However, we conjecture that much better results can be obtained by using subscript analysis to capture information about how an array is computed and how it is used.

6.1.5 Efficiency or Accuracy

In this thesis, we have tried to retain as much dependence information as possible. It might turn out that it is too expensive (at compile time) to retain all that information. It would then be worthwhile to trade off the accuracy of our algorithms to gain more efficiency.

Bibliography

[1] S. Abramsky. Strictness Analysis and Polymorphic Invariance. In *Programs as Data Objects, LNCS 217.* 1985.

[2] S. Abramsky and C. L. Hankin (eds). *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987.

[3] B. S. Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. CSG Memo, Laboratory for Computer Science, MIT, Cambridge, MA. Feb 1994.

[4] Z. Ariola and Arvind, P-TAC: A parallel intermediate language. In FPCA '89, pages 230-242, ACM, Sep 1989.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. In *Graph Reduction*, vol 279 of LNCS, pages 336-369, Springer-Verlag, Oct 1986.

[6] G. Baraki. A note on Abstract Interpretation of Polymorphic Functions. *In FPCA '91 (LNCS 513)*, pages 367–378. 1991.

[7] H. Barendregt. *The Lambda Calculus: Its Sytax and Semantics.* North-Holland, Amsterdam, 1984.

[8] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *FPCA '91*, vol 523 of *LNCS*, pages 538-568, Springer-Verlag, Aug 1991.

[9] A. Bloss. *Path Analysis and the Optimization of Non-strict Functional Languages.* PhD Thesis, Department of Computer Science, Yale University, May 1989.

[10] G. L. Burn, C. L. Hankin, and S. Abramsky. Strictness Analysis for Higher-order Functions. Science of Computer Programming, 7, 1986.

[11] C. Clack and S. L. Peyton Jones. Strictness Analysis – a practical approach. In *FPCA '85, LNCS 201.* Springer-Verlag. Sep 1985.

[12] T. H. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press. Cambridge, MA. 1990.

[13] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled thread abstract machine. In 4th ASPLOS, pages 164-175. ACM, Apr 1991.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[15] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-hall, Englewood Cliffs NJ, 1980.

[16] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele, Compile-time partitioning of a non-strict language into sequential threads, In *Proc. 3rd Symp. on Par. and Dist. Processing*, IEEE, Dec 1991.

[17] P. Hudak and P. Wadler (eds). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University Department of Computer Science, New Haven, CT, Apr 1990.

[18] P. Hudak and J. Young. Higher-order strictness analysis for the untyped lambda calculus. In *12th ACM Symposium on Principles of Programming Languages*, pages 97-109, Jan 1986.

[19] R. J. M. Hughes. Backwards analysis of functional programs. Research Report CSC/87/R3, University of Glasgow, Mar 1987.

[20] R. J. M. Hughes. Abstract Interpretation of First-Order Polymorphic Functions. Proceedings of the 1988 Glasgow Workshop on Functional Programming, Research Report 89/R4, University of Glasgow, 1989.

[21] R. A. Ianucci. *Parallel Machines: Parallel Machine Languages.* Kluwer Academic Publishers, Boston, 1990.

[22] T. Johnsson. Efficient compilation of lazy evaluation. *ACM SIGPLAN Notices*, 19(6):58–69, Jun 1984.

[23] T. Johnsson. Lambda Lifting. In *FPCA '85, LNCS 201*, pages 119-159, Berlin, October 1986. Springer-Verlag.

[24] J. Klop. Term Rewriting Systems. Course Notes, Summer course organized by Corrado Boehm, Ustica, Italy, Sep 1985.

[25] T. Kuo and P. Mishra. Strictness Analysis: A New Perspective based on Type Inference. *FPCA '89*, pages 260–272. 1989.

[26] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming (LNCS 83)*, pages 269-281, Berlin, Apr 1980. Springer-Verlag.

[27] R. S. Nikhil. Id version 90.0 reference manual. CSG Memo 284-1, MIT LCS, Cambridge MA, Sep 1990.

[28] R. S. Nikhil, G. M. Papadpoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19th Ann. Int. Symp. on Comp. Arch.* IEEE, May 1992.

[29] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token store architecture. In *Proc. 17th Ann. Int. Symp. on Comp. Arch.*, pages 82-91. IEEE, 1990.

[30] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proc. 18th An.. Int. Symp. on Comp. Arch.*, pages 342-351. IEEE, May 1991.

[31] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. In *Journal of Functional Programming*, **2** (2):127-202, Apr 1992.

[32] J. Rees and W. Clinger. Revised^3 report on the algorithmic languages scheme. Tech. Report. MIT-AI Lab, Cambridge, MA, 1986.

[33] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. TR-370, MIT LCS, Cambridge, MA, Aug 1986.

[34] K. R. Traub. Compilation as Partitioning: A new approach to compiling non-strict functional languages. In FPCA '89, pages 75-88, ACM, Sep 1989.

[35] K. R. Traub. *Implementation of Non-strict Functional Programming Languages.* MIT Press, Cambridge MA, 1991.

[36] K. R. Traub. Multi-thread code generation for dataflow architectures from non-strict programs. In FPCA '91, volume 523 of LNCS, pages 73-101, Springer-Verlag, Aug 1991.

[37] K. R. Traub, D. E. Culler, and K. E. Schauser. Global Analysis for Partitioning Non-Strict Programs into Sequential Threads. In *Proceedings of ACM-LFP*, San Francisco, Jun 1992.

[38] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In FPCA '91, volume 523 of LNCS, pages 73-101, Springer-Verlag, Aug 1991.

[39] D. A. Schmidt. *Denotational Semantics – A methodology for Language Development.* Allyn and Bacon, Inc., Boston, MA, 1986.

[40] Julian Seward. Polymorphic Strictness Analysis using Frontiers. In *FPCA '93.*

[41] D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, **9**:31–49. 1979.

[42] J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. *Journal of Computer and System Sciences,* 9(3):332-354, Dec 1974.

[43] P. Wadler. Strictness analysis on Non-Flat Domains by Abstract Interpretation. In [2].

[44] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *FPCA '87, LNCS 247*, pages 386-407, Berlin, Sep 1987. Springer-Verlag.

[45] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus.* PhD thesis, Oxford University, 1971.

[46] Y. Zhou. An Id Compiler in Id. Internal Memo. Computation Structures Group, MIT-LCS. Cambridge, MA. Sep 1992.