INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Semantic program dependence graphs

Parsons, Rebecca Jane, Ph.D.

Rice University, 1992
RICE UNIVERSITY

Semantic Program Dependence Graphs

by

Rebecca Parsons

A Thesis Submitted
in partial fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Matthias Felleisen, Co-Chairman
Assistant Professor of Computer Science

Robert S. Cartwright, Co-Chairman
Professor of Computer Science

Sydney Lamb
Professor of Linguistics and Semiotics

Houston, Texas

April, 1992
Semantic Program Dependence Graphs

Rebecca Parsons

Abstract

Semantic program dependence graphs, or semantic pdgs, are an attractive intermediate program representation for use in advanced optimizing and parallelizing compilers. The semantic pdg, which is based on the program dependence graph, has a compositional semantics that provides an elegant characterization of the types of dependences that arise in imperative programming languages. In addition, the semantic pdg has a simple operational semantics which serves as the basis of an equational calculus reasoning about semantic pdgs. Finally, the algorithms for creating the semantic pdg are efficient enough to allow the use of this program representation in actual compilers.

The semantic pdg is the result of a study, using denotational semantics, into the notions of data and control dependence in imperative programming languages. Semantic pdgs include a new component, the valve node, which ensures the data flow character of the semantic pdg, even in the presence of conditional assignments, and provides the control information necessary to perform many important program optimizations. The valve node is a natural result of the derivation step that addresses the data dependence relation. The semantic pdg utilizes the new concept of a partial array to allow for optimizations of array accesses while maintaining the mathematical elegance of the data flow semantics of the pdg.

The semantic pdg is not only an elegant representation from a mathematical perspective, but it is also a useful representation from a practical perspective. This structure is particularly well-suited for use in optimizing and parallelizing compilers since it explicates the important relationships among the different statements in a program. We have developed a program representation that is powerful enough to represent the behavior of a program, that provides the information needed to optimize the program, and that has a precise mathematical description. The development of the semantic pdg reconciles the often contradictory requirements of mathematical elegance and practicality.
Acknowledgments

I owe a debt of thanks to many people for making this whole experience possible. My advisor, Matthias Felleisen, introduced me to the world of language semantics that I have inhabited ever since. His advice, his proof-reading, and his support have been invaluable. I learned many things from Matthias, and I will continue to benefit from those lessons, even though the formal training period is over. My other advisor, Corky Cartwright, spent many hours listening to my ideas, talking them over with me, and giving me feedback. As with Matthias, I will continue to learn from Corky and benefit from his advice and support.

I would also like to thank the other member of my committee, Sydney Lamb, who did not look too startled at having a student appear out of nowhere to ask him to serve on her committee.

My thanks also go to my study companions and fellow torture chamber participants: Dorai, Rene’, Rama, Laura, Mike, Andrew, Amr, and John. Their help, advice and friendship enhanced my time here. To my friends across the country; Cheryl, Curt, Sue, Pat, Dennis, Don, Jan and Jim: thank you for helping me keep my sanity, and for giving me your friendship, your support, and many wonderful discussions. Mrs. Schmidt, I thank you as well. You did more to get me here than you know.

I would like to acknowledge the tremendous help Gina has given me over the years. She was volunteered to be my unofficial mentor when I first arrived, and little did she know what she was letting herself in for.

Finally, I would like to thank my parents, my sister and my brother. Without all of you, each in your own way, I wouldn’t be writing this now. Your support, your love, your friendship and your acceptance made this whole adventure possible.

I would like to acknowledge the financial assistance of IBM, through the Graduate Fellowship Program, Rice University and the National Science Foundation.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations viii

1 Introduction 1
1.1 Survey of Major Program Graph Representations 2
  1.1.1 Control Flow Graphs 2
  1.1.2 Data Flow Program Graphs 5
  1.1.3 Data Dependence Graphs 6
  1.1.4 Data-Flow Graphs 8
  1.1.5 Program Dependence Graphs 10
1.2 Towards a Semantic Development of PDG 11
1.3 Critique of the Program Dependence Graph 12
1.4 Outline 13

2 Definitions 15
2.1 Graph Theory 15
2.2 Control Flow Graph 17
2.3 Domain Theory and Denotational Semantics 19
  2.3.1 Domain Theory 20
  2.3.2 Domain Constructors 21
  2.3.3 Denotational Definitions 23
2.4 Program Dependence 24
2.5 Program Dependence Graphs 27

3 Semantics of Structured Program Dependence 29
3.1 Semantics for Language \( \mathcal{W} \) 29
  3.1.1 Denotational Semantics for Language \( \mathcal{W} \) 30
3.2 Dependence Trees for Language \( \mathcal{W} \) 32
3.2.1 Generalized Semantics for Language $W$ .................. 32
3.2.2 Data Dependence and Demand Semantics ................. 34
3.2.3 Control Dependence ........................................ 37
3.2.4 Program Dependence Trees .................................. 43
3.2.5 Example Program Dependence Tree ......................... 50
3.3 Operational Semantics for Program Dependence Trees ........ 51
3.3.1 Rewriting Example ......................................... 59
3.4 Soundness of Evaluation .................................... 64

4 Semantics of Dependence for Arrays ......................... 66
4.1 Program Dependences and Arrays .............................. 66
4.2 Language Semantics for Arrays ............................... 68
  4.2.1 Denotational Semantics of the Array Language .......... 68
4.3 Program Dependence Trees with Arrays ...................... 69
  4.3.1 Control Stage for Arrays ................................ 72
  4.3.2 Program Dependence Trees for Arrays ................. 74
  4.3.3 PDT Example with Arrays ................................ 83
4.4 Operational Semantics for Array PDTs ....................... 85
  4.4.1 Rewriting Example ..................................... 86

5 Semantics of Dependence for Abort ......................... 90
5.1 Language Semantics for Abort ............................... 90
5.2 Denotational Semantics for Abort ........................... 92
5.3 Control Dependence .......................................... 95
5.4 Program Dependence Trees in the Presence of Abort ....... 105
  5.4.1 Example .................................................. 119
5.5 Alternative View Of Control .................................. 120
5.6 Control Dependence in the Presence of Abort ................ 130
5.7 Operational Semantics for PDTs with Abort ................ 131
  5.7.1 Data Interpretation of Abort ............................ 131
  5.7.2 Control Interpretation of Abort ......................... 132
5.8 Discussion .................................................. 136

6 Semantics of Goto ........................................ 138
6.1 Language $W$ with General Control Flow .................. 138
6.1.1 Semantics of goto .................................. 139
6.1.2 Analysis ............................................. 141

6.2 Text Expansion Semantics for goto Statements .................................. 141
6.2.1 Dependence Under Text Expansion ............................................. 144

6.3 Lazy Semantics for Goto Statements ............................................. 146
6.3.1 Derivation of the Post-Dominator Relation .................................. 146
6.3.2 Control Dependence ............................................. 152
6.3.3 Program Dependence Trees ............................................. 159

6.4 Operational Semantics in the Presence of GOTO .................................. 166

6.5 Example ............................................. 166
6.6 Discussion ............................................. 168

7 Implementation Issues ............................................. 169
7.1 Collapsing PDTs to Semantic PDGs ............................................. 170
7.2 Characteristics of the Semantic $P_{dg}$ ............................................. 174

7.3 Semantic PDG Algorithm ............................................. 176
7.4 Examples of Semantic PDGs ............................................. 180

8 Semantic Program Dependence Graphs ............................................. 183
8.1 Issues in Program Dependence ............................................. 184
8.1.1 Store Filtering ............................................. 184
8.1.2 Lazy Semantics and Parallelism ............................................. 185
8.1.3 Lexical Scoping ............................................. 186
8.1.4 Procedures ............................................. 187
8.1.5 Pointers ............................................. 188
8.1.6 Demand-Driven and Data-Driven Semantics ............................................. 188
8.1.7 Differences between abort and goto ............................................. 189
8.1.8 Applicability to Optimizations ............................................. 190

8.2 Other Dependence-Based Graph Structures and Semantic Analyses of
Optimization ............................................. 191
8.2.1 Comparison of the Different Representations ............................................. 194

8.3 Future Work ............................................. 195
8.3.1 Language Enhancements ............................................. 195
8.3.2 Additional Semantic Development and the Calculus ............................................. 196
8.3.3 Optimizations and Implementations ............................................. 196
8.3.4 Dynamic Dependence ......................... 197
8.4 Semantics of Generalized Program Dependence .......... 198

Bibliography ........................................... 199
# Illustrations

1.1 Example Program .................................................. 3  
1.2 Control Flow Graph for Example Program ..................... 4  
1.3 Data Flow Program Graph of Example Program ................. 6  
1.4 Data Dependence Graph for Example Program ................. 7  
1.5 Extended Data Flow Graph for Example Program .............. 9  
1.6 Pdg for Example Program ......................................... 11  

2.1 Sample Directed Cyclic Graph ...................................... 16  
2.2 An Irreducible Control Flow Graph .............................. 19  

3.1 Definition for \(A\) .................................................. 30  
3.2 Denotational Definition of Language \(W\) .......................... 31  
3.3 Definition of the Strict Store Update Function ............... 31  
3.4 Demand Semantics for Language \(W\) ............................. 35  
3.5 Control Semantics for Language \(W\) ............................. 38  
3.6 Abstraction Step .................................................. 45  
3.7 Domains for Program Dependence Trees ......................... 46  
3.8 Semantics for Program Dependence Trees ....................... 47  
3.9 Program Dependence Tree for Program \(P\) ....................... 50  
3.10 Mapping Function from Domain Element to Graph Node ....... 52  
3.11 Representations for PDT Nodes .................................. 53  
3.12 Rewriting Rules for the Core Language ........................ 56  
3.13 Substitution Operation for Nodes ............................... 56  
3.14 \(Pdt\) Graph for Program \(Q\) .................................. 60  
3.15 \(Pdt\) Graph for Program \(Q\) After 1 Rewriting Step .......... 61  
3.16 \(Pdt\) Graph for Program \(Q\) After 2 Rewriting Steps .......... 62  
3.17 \(Pdt\) Graph for Program \(Q\) After 3 Rewriting Steps .......... 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.18</td>
<td>Pdt Graph for Program Q After 4 Rewriting Steps</td>
<td>63</td>
</tr>
<tr>
<td>3.19</td>
<td>Final Pdt for Program Q</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Denotational Definition of the Array Language</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Data Stage for Array Semantics</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>Control Stage of Array Derivation</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>Definitions for O(A)</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Abstraction Stage of Array Semantics</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Merge Node Semantics for Arrays</td>
<td>76</td>
</tr>
<tr>
<td>4.7</td>
<td>Help Functions for Array Semantics</td>
<td>77</td>
</tr>
<tr>
<td>4.8</td>
<td>Array Merge Node Example</td>
<td>84</td>
</tr>
<tr>
<td>4.9</td>
<td>Representations for PDT Nodes</td>
<td>86</td>
</tr>
<tr>
<td>4.10</td>
<td>Rewriting Rules for the Array PDTs</td>
<td>87</td>
</tr>
<tr>
<td>4.11</td>
<td>Array Rewriting Example - Original Program</td>
<td>88</td>
</tr>
<tr>
<td>4.12</td>
<td>Array Rewriting Example - Predicates Removed</td>
<td>88</td>
</tr>
<tr>
<td>4.13</td>
<td>Array Rewriting Example - First Merge Node</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Denotational Semantics for Abort</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Data Stage for Abort Semantics</td>
<td>92</td>
</tr>
<tr>
<td>5.3</td>
<td>Domains for the Control Stage of the Abort Semantics</td>
<td>95</td>
</tr>
<tr>
<td>5.4</td>
<td>Control Stage for Abort Semantics</td>
<td>96</td>
</tr>
<tr>
<td>5.5</td>
<td>Definition for A(B)</td>
<td>98</td>
</tr>
<tr>
<td>5.6</td>
<td>Domains for the Abstraction Stage of Abort Semantics</td>
<td>105</td>
</tr>
<tr>
<td>5.7</td>
<td>Meaning Functions for the Abstraction Stage of Abort Semantics</td>
<td>106</td>
</tr>
<tr>
<td>5.8</td>
<td>Domains for PDG Functions</td>
<td>108</td>
</tr>
<tr>
<td>5.9</td>
<td>G Function for the Abort Semantics</td>
<td>109</td>
</tr>
<tr>
<td>5.10</td>
<td>Final Stage for Abort Semantics</td>
<td>111</td>
</tr>
<tr>
<td>5.11</td>
<td>Example abort Program under Control Perspective</td>
<td>119</td>
</tr>
<tr>
<td>5.12</td>
<td>Example abort Program under Traditional Dependence Definitions</td>
<td>121</td>
</tr>
<tr>
<td>5.13</td>
<td>Example abort Program under Data Perspective</td>
<td>121</td>
</tr>
<tr>
<td>5.14</td>
<td>Domains for Data Flow Interpretation of Abort</td>
<td>122</td>
</tr>
<tr>
<td>5.15</td>
<td>Data Flow Interpretation of Abort</td>
<td>123</td>
</tr>
<tr>
<td>5.16</td>
<td>Data Flow Interpretation of Abort (cont)</td>
<td>124</td>
</tr>
<tr>
<td>5.17</td>
<td>Example for Control Dependence and abort</td>
<td>131</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Programming tools that manipulate programs, such as compilers and programming environments, generally use an intermediate representation for the program instead of the program text. These intermediate representations are designed to retain the important information about the source program, make this information readily accessible to the tool processing the information, and hide the irrelevant information about the source program. An important intermediate representation for optimizing compilers is the program dependence graph, or \textit{pdg}. The \textit{pdg} is an intuitively appealing representation that eliminates the sequencing constraints imposed by the sequencing operations of traditional sequential programming languages. Instead, \textit{dependencies} in the program constrain the sequencing of statements. Intuitively, a dependence exists between two statements when the correct processing of the second statement requires that the first statement execute prior to the execution of the second statement.

The \textit{pdg} is used in scalar optimizing compilers [6, 20, 45, 59], programming environments [36], and program integration systems [30], as well as parallelizing [23] and vectorizing [10] compilers. In each of these applications, part of the information that must be retained by the representation is the meaning of the program. A semantics for a particular \textit{pdg} representation is a mapping from instances of the representation to their meanings. Given a semantics for \textit{pdgs}, a semantics for programs and a transformation from programs to their corresponding \textit{pdgs}, the meanings of \textit{pdgs} and programs can be compared and statements can be made about the adequacy of the \textit{pdg} as an intermediate representation. A representation is \textit{adequate} if the meaning of a program under the programming language semantics corresponds to the meaning of the instance of the representation for that program under the semantics for the representation. Since programming tools manipulate \textit{pdgs} instead of program text, the correctness of the tools depends on the the effects of the transformations on \textit{pdgs} and is thus most conveniently formulated in these terms. A semantics for \textit{pdgs} provides a mechanism for examining the correctness of such tools, without appealing to the semantics of program text and the translation between program text and \textit{pdgs}. 
A semantics for pdgs also allows pdgs to be studied independently of a particular programming language. Indeed, pdgs are programs in a pdg programming language, supporting a parallel model of computation. A semantics for pdgs also provides the basis for other semantic tools, such as equational calculi [9, 21, 42], which are directly applicable to pdgs, without reverting back to the semantics of the original programming language. These calculi are useful in establishing the validity of transformations by allowing proofs of the equivalence of non-isomorphic pdgs. The equality of the pdg of the original program and the pdg for the transformed program establishes the validity of the transformation.

This dissertation develops a semantics foundation for pdgs and program dependence. The denotational semantics and operational semantics for pdgs provide meanings for pdgs representing programs in general imperative languages. These semantics provide a mathematical foundation for pdgs and allow a precise understanding of the different dependence relations. The framework provides the tools necessary to formally prove properties of pdgs; the semantic definitions specify the semantic information about pdgs that is required to build these tools.

1.1 Survey of Major Program Graph Representations

Programming language researchers have developed several different types of graph structures to represent programs. Each type identifies certain relationships while leaving other information implicit. In this section, we survey the important program representations, emphasizing those that have contributed to the development of the pdg. The program appearing in Figure 1.1 serves as the example program for this section. This program is shown using each of the different representations discussed in this section.

1.1.1 Control Flow Graphs

The most basic form of graph representation is the control flow graph [2], which is essentially an augmentation of the flowchart [58] of a program. These graphs are an abstraction of the machine code of von Neumann computers. The informal semantics of control flow graphs derives from the correspondence between these graphs and machine code. Conceptually, the meaning of a control flow graph is function mapping stores to stores. Conceptually, each node in the control flow graph uses the store it receives from some input edge and passes a potentially modified store along one of its
\begin{verbatim}
while $x \leq 10$
  $x := x + 1;$
if $y \leq 42$
  goto 4
  goto 5;
4 : $y := x \times y;$
5 : $z := x + y;$
end
\end{verbatim}

**Figure 1.1** Example Program

output edges. Predicate nodes pass the store along one of the possible edges, while other nodes update the store before passing it on to subsequent nodes.

In the control flow graph for a program, which is a directed, potentially cyclic graph, there is a **start** node, an **end** node, predicate nodes and assignment nodes. The **start** node has no predecessors, and the **end** node has no successors. Predicate nodes have two or more successors while assignment nodes have only one successor. Every node lies on a path from the **start** node to the **end** node. The predicate and assignment nodes may be single statements or multiple statements. Predicate nodes have a predicate as their last component. The value of this predicate determines the choice of path from the node. Cycles in the control flow graph represent loops in the corresponding program. The edges in the control flow graph are partitioned into forward and backward edges. Each loop has at least one backward edge associated with it. The control flow graph contains no explicit dependence information. Instead, the edges in the graph sequence the evaluation based on the textual position of the statements in the program and not with regard to the way in which the different statements interact during program evaluation. Thus, although there are no explicit dependences, the evaluation of the program is over-constrained by the sequencing edges of the control flow graph.

The control flow graph for the example program appears in Figure 1.2. The example control flow graph includes the distinguished nodes **start** and **end**. The nodes $x \leq 10$ and $y \leq 42$ are predicate nodes; all other nodes are assignment nodes. The edge from the assignment to $x$ to the predicate $x \leq 10$ is a back edge forming the loop. The labels on the edges from the predicate nodes designate under which
Figure 1.2  Control Flow Graph for Example Program
conditions these paths are followed, where $T$ designates the path taken if the predicate is $true$ and $F$ is the $false$ path.

1.1.2 Data Flow Program Graphs

While the edges in the control flow graph represent the flow of control, edges in data flow program graphs [55] represent the flow of data in the program. Data flow programs contain flow edges that transmit values from the node producing the value to the nodes using the value. The execution of a data flow program proceeds by evaluating nodes once all of their inputs are available. Conditional control flow is included in data flow program graphs by gating the inputs into the affected nodes. Switches perform this gating function by accepting a predicate value and a data value. A switch propagates the data value based on the predicate value. Thus, control flow information is implicit in data flow graphs. Data flow graphs are potentially cyclic to allow loops in the programs.

Like the control flow graph, the data flow graph is a low-level programming language. In contrast, the semantics of control flow graphs, there is no central store in the data flow model of computation. Data values flow over the edges in the data flow program graph to the nodes using the values, distributing the store across all the nodes in the data flow program graph.

Kosinski [34] specified the semantics of data flow graphs as systems of simultaneous equations. The computational model for data flow program graphs is parallel, asynchronous, and in general non-deterministic, in contrast to the sequential computational model of the control flow graphs. Conceptually, a control flow graph can be thought of as a data flow program graph where the store is the value that flows over the edges in the graph. However, the edges still force the sequential computational style that is commonly ascribed to the control flow graph.

A data flow program graph for the example program appears in Figure 1.3. The nodes labeled $x$ and $y$ represent the input values for these identifiers. There are switches gating the values for $x$ and $y$ flowing into the assignment node, $y := x \cdot y$. Values will not flow into this node unless the predicate $y \leq 42$ is $true$. The evaluation of this assignment to $y$ is disabled if the predicate is $false$, since the required input values will never arrive. The switches encode the control information in the data flow graph. The data flow program graph also includes a fair, non-deterministic data flow merge node, labeled $M$. This merge node non-deterministically selects one of
Figure 1.3  Data Flow Program Graph of Example Program

its inputs to propagate as values arrive over its different input edges. The fairness criteria ensures that any value that arrives must eventually be sent on to the output of the merge. The merge nodes allow for the proper flow and merging of values from inside and outside a loop.

1.1.3 Data Dependence Graphs

Kuck [35] first specified the notion of data dependence, based on Bernstein’s conditions [12] concerning the safety of executing statements concurrently. Kuck identified two different types of data dependence: flow dependence (called true dependence) and storage dependence. A flow dependence relates an assignment to an identifier to a use of that identifier. Storage dependences are sub-divided into anti-dependence, input dependence and output dependence. Anti-dependences sequence the use of an identifier before a subsequent store to the same identifier to ensure that a use of that identifier does not get an updated value for the identifier when the original value is needed. Input dependences connect uses of the same identifier and are included for completeness only and are generally only used to optimize memory accesses. Output
dependence sequence assignments to the same identifier so that the value of an identifier is the most recent value.

Combining data dependence information with control flow information of the control flow graph introduces some of the parallel character that is inherent in the data flow graph into the control flow graph. At this point in the development of the representations, however, the data dependence information is maintained separately from the control information, in a structure called the *data dependence graph*. The data dependences are labeled edges in the data dependence graph. Nodes in the data dependence graph are generally either statements or block of statements. In the statement level data dependence graph, nodes are either assignment statements, *if* statements or *while* statements. Data dependence graphs only partially represent programs since no control flow or control dependence information is present.

Allen and Kennedy [5] refined the notions of data dependence in the presence of loops and array identifiers, partitioning data dependences into loop-carried and loop-independent dependences. In addition, the nesting level (or *depth*) of the loop that *carries* the dependence is included in the label of a loop-carried dependence. This information is useful in optimizations for vector architectures.

Figure 1.4 presents the data dependence for the example program. There is an anti-dependence from the *while* predicate to the assignment to *x* within the loop, flow dependences from the assignment in the loop to each of the assignments outside

```
\begin{center}
\begin{tikzpicture}
  \node (y) at (0,0) {$y \leq 42$};
  \node (x10) at (-2,-1) {$x \leq 10$};
  \node (c) at (-3,-2) {$c$};
  \node (x) at (0,-2) {$x := x + 1$};
  \node (y) at (1,-2) {$y := x \times y$};
  \node (z) at (2,-2) {$z := x + y$};

  \draw[->] (y) -- (x10);
  \draw[->] (c) -- (x10);
  \draw[->] (x10) -- (x);
  \draw[->] (y) -- (x);
  \draw[->] (z) -- (x);
  \draw[->] (y) -- (z);

\end{tikzpicture}
\end{center}
```

**Figure 1.4** Data Dependence Graph for Example Program
the loop, and a flow dependence from the assignment to \( y \) to the assignment to \( z \). There is a loop-carried flow dependence from the assignment to \( x \) back to the \textbf{while} predicate. Finally, there is a loop-carried flow dependence, a loop-carried output dependence and a loop-carried anti-dependence from the assignment to \( x \) back to itself.

Several approaches for accommodating control flow have been proposed. The first technique, if-conversion [4], translates control flow information into data dependences by using guards on statements. Under if-conversion, all loop-exit branches and forward branches are replaced with \textbf{if} statements and boolean guards on the affected statements. Backward branches are translated into \textbf{while} loops. Thus, the only control structures are \textbf{while} loops and \textbf{if} statements that set the guards. While this translation does include the control information in the graph structure, the distinction between control and data dependence is lost. In addition, the translation significantly alters the structure of the program.

1.1.4 Data-Flow Graphs

Ottenstein [37] proposed joining control information and data dependence information by combining the control flow graph with the data dependence graph. This combined structure, called the data-flow graph, or \( dfg \), is the predecessor to the \( pdg \) [24]. The \( dfg \) contains flow edges connecting the use of an identifier with nodes assigning a value for that use of the identifier, or to a special value, \textit{undefined}. None of the other data or control dependences appear in the structure. Instead, each node in the \( dfg \) contains a pointer to the corresponding node in the control flow graph. This mapping provides the control flow information to allow some optimizing transformations on the structure. The name of this structure is misleading since it is not a data flow program graph. However, the inspiration for the \( dfg \) did come from the data flow model of computation.

Ottenstein did not present a formal semantics for the \( dfg \). Because the conditions under which statements are enabled for evaluation, the control dependence relation for the program, is still represented as control flow information and is thus implicit, the semantics for the \( dfg \) is formulated most readily using the semantics for the control flow graph. The data dependence information is thus treated as an additional annotation that can be exploited in the optimization process but has no effect on the meaning of the program.
The $dfg$ was extended by Ferrante and Ottenstein [25] to include control dependences explicitly, instead of using the control flow information. However, this new structure, the extended data flow graph or $edfg$, is defined only for structured programs with repeat statements$^1$ where the control dependences reflect the nesting structure of the programs. The control dependences defined here do not correspond exactly to control dependences as currently defined and used in the literature; however, the spirit of the definitions are the same. The $edfg$ for the example program appears in Figure 1.5.

With the explicit control dependence information in the $edfg$, the semantics of data flow program graphs can serve as an informal semantics for the $edfg$. There is no central store in this model, and the mapping from the program to the $edfg$ separates conflicting uses of identifier names. Merge nodes are used to join the flow of values from multiple sources. These merge nodes act as local stores for conditionally assigned values. Using a data flow style semantics, the rest of the store is distributed among the nodes in the $edfg$.

---

**Figure 1.5** Extended Data Flow Graph for Example Program

$^1$We define structured programs as programs with if, and while control constructs in addition to assignment statements and a statement composition operator.
1.1.5 Program Dependence Graphs

The pdg [24], developed by Ferrante, Ottenstein and Warren, contains both the data dependence information and the control dependence information for programs with general control flow. The control information is included using the notion of a control dependence as opposed to the control flow graph edges or the simpler notion of control dependence used previously. Intuitively, a control dependence exists whenever the outcome of a predicate may determine whether or not some statement is executed. This explanation actually describes the set of all control dependence predecessors, while the actual control dependence relation is restricted to the immediate control dependence predecessor. Control dependence edges are labeled with a boolean value corresponding to the predicate value under which the node is enabled for evaluation. In structured programs, the control dependence relation reflects the nesting structure of the program. In unstructured programs, the control dependence relation is more complex. If node n is control dependent on node m with the label true and the predicate expression in node m is true, then node n must execute. In terms of the control flow graph, the control flow of the program must pass through the node n. If the predicate expression in m evaluates to false, then node n may or may not execute, depending on the other control dependences for n. The control dependence relation defined for pdgs differs from that used in edfgs, even for structured programs. In the pdg the predicate nodes for while loops and repeat loops are control dependent on themselves; these edges are implicit in the edfg. In addition, in the pdg for a program with nested repeat loops, the body of the nested loops are control dependent on all repeat predicates; the control dependences in the edfg reflect only the nesting structure, so each node has a single control dependence predecessor.

The pdg as defined by Ferrante et al. [24] includes output dependence edges and anti-dependence edges in addition to the flow dependences edges in the dfg and edfg. Anti- and output dependences are storage dependences that arise from the reuse of identifier names and conditional assignments of identifiers when identifier values are retrieved from a central store. The dfg implicitly renames all identifiers and merges multiple assignments for a given use of an identifier by distributing the store to the individual nodes, so these additional dependences are not required.

The pdg for the example program appears in Figure 1.6. The output and anti-dependence edges are labeled with O and A respectively.
1.2 Towards a Semantic Development of PDG

The notions of data dependence and control dependence are natural and intuitive. Unfortunately, this intuitive understanding inhibited a solid semantic development for the pdg. However, intuition and informal semantics are inadequate for a complete understanding of the pdg as a program representation.

Horwitz et al. [31] first addressed the issue of the theoretical foundations of pdgs by addressing the adequacy of the pdg as a program representation. While they did not specify a semantics for pdgs, they did show a Weak Adequacy Theorem for pdgs for structured programs with scalar identifiers using def-order dependence instead of output dependence:

**Theorem 1**  Weak Adequacy Theorem for Structured Programs

Let $P$ and $Q$ be structured programs with isomorphic pdgs $G_P$ and $G_Q$. Then for any store $\sigma$,

1. $P$ and $Q$ both diverge for $\sigma$, or
2. $P$ and $Q$ both terminate for $\sigma$ and all final values for identifiers are the same

This theorem shows that different programs with the same pdg must have the same behavior. However, this theorem does not give a meaning to the pdg or allow state-
ments to be made about pdgs that are not isomorphic. In addition, the theorem does not provide insight into the nature of dependence, even for the restricted language. No attempt is made to extend the language to more general control constructs or to handle complex data values. Finally, although the report states that the Weak Adequacy Theorem shows the viability of a direct interpretation of the pdg, no interpretation is given. In fact, we describe in Section 1.3 later in this report that a fully parallel interpretation requires a slightly different specification of the pdg.

Pdgs are similar to data flow graphs [55]. Semantics for data flow graphs are generally presented as sets of simultaneous equations [34]. Once again, this semantic presentation does not give insight into the nature of dependence. In addition, data flow graphs do not have a direct analogue for control dependences; thus, a translation of control dependences to data flow switches would have to be performed before applying the data flow program semantics. Finally, the flow of information in the pdgs defined above is not pure data flow. With either def-order dependence or output dependence, some internal state is required to store the multiple values that may arrive for an identifier at a node. Without this internal state and some mechanism for tracking the control flow of the program, no semantics can be given to the pdg.

1.3 Critique of the Program Dependence Graph

The pdg is a useful program representation for programming tools and environments. However, the specification of the pdg presented in the previous section has several shortcomings. First, the pdg was introduced without any semantic definitions. The informal semantics for the pdg was the data flow model of computation. However, the pdg does not fulfill the criteria for a data flow program. The pdg assumes the presence of a central store, which is not a data flow concept. While renaming of identifiers [19] resolves some of the issues surrounding the use of a central store in the pdg, the problem of conditional assignments to identifiers remains. The def-order dependence pdg proposed by Horwitz et al. [31] addresses this problem. Unfortunately, def-order dependence does not allow for a compositional semantics. Thus, the pdgs proposed in the literature are not semantically sound representations.

Second, the operational semantics developed by the author [48] uncovered a shortcoming in the definition of the def-order pdg. The def-order pdg is not defined with loop-carried def-order edges. However, in order to properly assign a parallel semantics to the pdg that allows statements for different loop iterations to execute concurrently,
these edges are required. While current uses of the $pdg$ do not require this capability, the discovery of this problem highlights both the inadequacy of informal methods in understanding the $pdg$ and program dependences, and the usefulness of the tools of semantics in studying these issues.

1.4 Outline

In this dissertation, we study the semantics of dependence representations for a general imperative programming language. Both the dependence representation and the corresponding semantics can be formally derived from conventional denotational definitions. They provide a rigorous mathematical foundation for transforming and analyzing programs. The dependence representation resulting from this derivation identifies areas of improvement in the traditional specification of the $pdg$. In addition, the derivation suggests new approaches to removing dependences; these approaches appear to be useful in improving the optimization process.

The preceding sections surveyed the historical development of the program dependence graph, highlighted the lack of semantic development for program dependence and the $pdg$ and described the shortcomings of the current definitions of the $pdg$. This dissertation provides a semantic framework for $pdgs$ which supports structured control operations, general control operations in the form of $goto$ statements, and structured data values in the form of arrays, and addresses the shortcomings of the current specification. The language considered provides programming language features found in the bulk of imperative programs targeted for parallel architectures. The primary limitation of the language is the absence of procedures and pointers. These limitations are discussed in Section 8.1 in the final chapter.

The semantic specification for $pdgs$ discussed here includes both denotational and operational characterizations. The denotational derivations expose the nature of the dependences in the $pdg$ while the operational semantics codifies the programmer’s notion of computation as it relates to the evaluation of parallel programs. Chapter 2 formally defines the $pdg$ and the different types of dependences discussed in this chapter, in addition to providing the background information required to understand the rest of the dissertation. Chapter 3 reviews the work of Cartwright and Felleisen [15, 14] and Selke [48] which provides a semantic foundation for dependence in structured programs with atomic data values. Chapter 4 adds structured data values in the form of array values. The denotational derivation incorporates an oracle that captures the
subscript analysis performed by advanced compilers and adds a new node type to the \textit{pdg}, the \textbf{merge} node. The \textbf{merge} node combines the previous views of arrays as memory locations and arrays as functional objects to increase the amount of parallelism available. Chapter 5 examines the impact of structured control mechanisms on control dependence. Two characterizations of \texttt{abort}, one from a data perspective and one from a control perspective, are presented. Once again, both a denotational and an operational description is provided. Chapter 6 incorporates general control operations into the language. The derivation process used in the preceding chapters breaks down in the presence of general control flow. Two approaches are described here, specifying two different control dependence relations. Chapter 7 gives a \textit{practical} algorithm for creating the \textit{pdg} defined in the preceding chapters. The correctness and complexity of the algorithm are both established. In Chapter 8, we place our work in perspective, examine other attempts at improving the \textit{pdg}, and discuss future work.
Chapter 2

Definitions

Understanding the material in this dissertation requires some fundamental concepts in graph theory, domain theory and denotational semantics. This chapter presents the technical definitions that lay the foundation for this material. The first section defines some graph theoretic concepts underlying the various graph representations. The second section describes terminology commonly associated with control flow graphs. Section 2.3 defines the concepts from domain theory required to understand the denotational definitions. The fourth section defines the different notions of dependence presented in the literature. The final section describes the pdg [24].

2.1 Graph Theory

Pdg.s and control flow graphs are directed graphs [1].

**Definition 2.1 (Directed Graph)** A directed graph \( G = (N, E) \) consists of a node set \( N \) and an edge set \( E \subseteq N \times N \). An edge \( e = (n, m) \in E \) is an ordered pair of nodes \( n, m \in N \). The *source* of the edge \( (n, m) \) is node \( n \), and the *target* of the edge is node \( m \). Node \( m \) is a *successor* of node \( n \), and node \( n \) is a *predecessor* of node \( m \). A *source* node is a node \( n \) such that there is no edge \( (m, n) \in E \) for any \( m \in N \). A *sink* node is a node \( n \) such that there is no edge \( (n, m) \in E \) for any \( m \in N \).

Figure 2.1 shows a sample directed graph with node set \( N = \{A, B, C\} \). The edge set is \( E = \{(A, B), (B, A), (B, C), (C, A)\} \). In other words, there are edges from \( A \) to \( B \), from \( B \) to \( A \), from \( B \) to \( C \), and from \( C \) to \( A \).

Alternatively, a graph \( G = (N, S) \) can be characterized with a successor relation \( S \) such that

\[(n, m) \in E \text{ iff } m \in S(n)\]

---

2Since the edge set \( E \subseteq N \times N \), there can be only one edge \( (n, m) \) in a graph. Since the graphs are directed, however, the edge \( (n, m) \) is distinct from the edge \( (m, n) \). A *multi-graph* can contain multiple edges between two nodes.
for all edges in the edge set $E$. A node may also have additional information associated with it. Both the node set and the edge set may be partitioned to distinguish between nodes or edges of different types.

Many of the dependence definitions require the notion of a path in a directed graph.

**Definition 2.2 (Directed Path)** A path $p$ in a graph $G = (N, E)$ from node $n_1$ to node $n_k$ is a sequence of edges of the form $(n_1, n_2), \ldots, (n_{k-1}, n_k)$ for $n_i \in N, 1 \leq i \leq k$ and $(n_{j-1}, n_j) \in E, 2 \leq j \leq k$; the specified path is of length $k - 1$ and each node $n_i$ is on path $p$. A path is *simple* if all edges and all nodes on the path, except the first and last nodes, are distinct. A *cycle* is a simple path of length at least one that begins and ends at the same node. A graph is *cyclic* if it contains at least one cycle; otherwise it is *acyclic*.

In Figure 2.1, the path $(A, B), (B, C)$ is of length two. There are two cycles: one from $A$ to $B$ and back to $A$, and one from $A$ to $B$ to $C$ and then back to $A$.

Algorithms on control flow graphs frequently use the notion of a connected component [1] to characterize loops in a program.

**Definition 2.3 (Strongly Connected Component and Weakly Connected Component)** Let $G = (N, E)$ be a directed graph. A strongly connected component in $G$ is a set of nodes $n_i \in N, 1 \leq i \leq r$ such that there is a path in $G$ from each node $n_i$ to every other node $n_j, 1 \leq j \leq r$. A weakly connected component in $G$ is a set of nodes $n_i \in N, 1 \leq i \leq r$ such that for each pair of nodes $n_i$ and $n_j$ there is either a path in $G$ from $n_i$ to $n_j$ or a path in $G$ from $n_j$ to $n_i$. 

---

**Figure 2.1** Sample Directed Cyclic Graph

- **A**
- **B**
- **C**
The dominator [2] and post-dominator [20, 24] relations for a graph are used in the definitions of dependences and control flow graphs.

**Definition 2.4 (Dominator and Post-Dominator in a Graph)** Let $G = (N, E)$ be a directed graph. Node $n$ dominates node $m$ with respect to node $n'$ if for all paths $p$ from node $n'$ to node $m$, $n$ is on the path $p$. Node $n$ strictly dominates node $m$ if $n$ dominates $m$ and $n \neq m$. Node $n$ (strictly) dominates node $m$ if it (strictly) dominates $m$ with respect to the start node. Node $n$ post-dominates node $m$ with respect to node $n'$ if for all paths $p$ from node $m$ to node $n'$, $n$ is on the path $p$. Node $n$ strictly post-dominates node $m$ if $n$ post-dominates $m$ and $n \neq m$. Node $n$ (strictly) post-dominates node $m$ if it (strictly) post-dominates $m$ with respect to the end node. \(^3\)

A subset of directed graphs are trees.

**Definition 2.5 (Trees and Forests)** A directed graph $G = (N, E)$ is a tree if each node in $G$ has only one predecessor and if there is a node $n$ such that all nodes are on some path from $n$; $n$ is the root of the tree. A forest is a graph that is a set of trees.

### 2.2 Control Flow Graph

This section formally defines control flow graphs [2]. Many of the dependence relations are defined in terms of the control flow graph.

**Definition 2.6 (Control Flow Graphs)** Let $Exp$ be a set of expressions and $Id$ a set of identifiers, deliberately unspecified. A control flow graph is a directed graph $C_P = (C_N, C_E)$ where

$$C_N = Exp + (Id \times Exp) + \{\text{start}, \text{end}\}$$

and

$$C_E \subset C_N \times C_N$$

\(^3\)The definition of post-dominance given by Ferrante *et al.* [24] is irreflexive. Cytron *et al.* [20] present the definition used here.
The nodes in the node set $C_N$ are either the designated start node, the designated end node, or are partitioned into two classes of nodes: predicate nodes, members of the set $Exp$, and assignment nodes, members of the set $Id \times Exp$. The start is a source node in the graph, and the end node is a sink node in the graph. Assignment nodes have a single successor while predicate nodes have two successors, labeled T and F. The start node has one successor. All nodes are on some path from the start node to the end node.

Control flow graph nodes are expressions, or individual program statements. Predicate nodes correspond to conditional control statements in a program; assignment nodes correspond to assignment statements. Edges in the control flow graph represent the flow of control.

The mapping between a program and its control flow graph is a natural one [2]. Each statement in a program is a node in the graph. There is an edge from the start node to the node corresponding to the first statement in the program. There is a directed edge from node $m$ to node $n$ if the statement for node $n$ immediately follows the statement for node $m$ on some evaluation path in the program. The label on the edges from a predicate node represent the value of the predicate expression that corresponds to this path in the program. Any path that in the program that terminates at the end of the program terminates at the end node in the control flow graph.

The control flow graph of a program is an augmented flowchart [58] version of the program. The start and end nodes are added to the flowchart of the program, along with the additional edges from the start node and the edge to the end node. These additional nodes simplify the definitions of some of the dependences.

Control flow graphs are characterized as reducible or irreducible [2].

**Definition 2.7 (Reducible and Irreducible Control Flow Graphs)** A control flow graph $C_G = (C_N, C_E)$ is reducible if the set of edges $C_E$ can be partitioned into forward and backward edges with the following properties:

1. The forward edges form an acyclic graph in which every node can be reached from the start node in $C_N$.

---

4The nodes in a control flow graph are sometimes sequences of statements called basic blocks [2], but these structures are not relevant to this dissertation.
2. For each backward edge \((n, m)\), \(n\) dominates \(m\) with respect to the start node.

If a control flow graph is not reducible, it is irreducible.

Semantically, the control flow graph for a program is reducible if all loops in the program are single-entry. Each backward edge in a reducible control flow graph identifies a loop in the program. The statements in the loop form a strongly connected component in the control flow graph. Figure 2.2 shows an irreducible control flow graph. This graph has a cycle with nodes \(B\) and \(C\) which can be entered either along the path from \(A\) to \(B\) or on the path from \(A\) to \(C\). Certain data flow analysis algorithms do not work on irreducible control flow graphs. However, node splitting [2] converts an irreducible control flow graph to a semantically equivalent reducible control flow graph.

### 2.3 Domain Theory and Denotational Semantics

A denotational definition specifies the meaning of a programming language in terms of a collection of semantic domains and a meaning function. The meaning function

![Diagram]

**Figure 2.2** An Irreducible Control Flow Graph
maps programs in the language to some element in the appropriate semantic domain in a compositional manner. Specifically, the meaning of a program component is a function of the meaning of its component parts. For imperative languages, the domain of meanings typically is some form of store transformer; the meaning of a program is a function from an input store to an output store. The operations on the store specify the meanings of the different components of the language. This section reviews domain theory, the mathematical theory describing the semantic domains, and denotational definitions.

2.3.1 Domain Theory

Domain theory provides the mathematical framework for denotational semantics. The theory specifies the structure of semantic domains; these structures ensure that the meaning functions in denotational definitions, if constructed properly, exist and are well-defined. While a complete treatment of domain theory is beyond our scope (see [47, 46, 51] for complete details), the basic concepts are described in this section, along with the specific domain constructions used in the subsequent chapters.

Intuitively, domain theory presents a model of computation where input and output occur incrementally, with the constraint that new pieces of information are consistent with the previous pieces. A partial ordering relation, referred to as approximation, defines the notion of consistency.

Specifically, a domain is a complete partial order [51].

Definition 2.8 (Complete Partial Order) A partial order \( B, \sqsubseteq \) is a pair \( (B, \sqsubseteq) \) where \( \sqsubseteq \) is a reflexive, transitive, antisymmetric binary relation on the set \( B \). The least upper bound, abbreviated lub, for two elements \( x \) and \( y \), denoted \( x \sqcup y \), in a partial order \( (B, \sqsubseteq) \), is an element \( z \in B \) such that

\[
x \sqsubseteq z, y \sqsubseteq z, \text{ and } \forall b \in B. x \sqsubseteq b \land y \sqsubseteq b \supseteq z \sqsubseteq b
\]

The lub of a set \( S \subseteq B \), denoted \( \sqcup S \) is an element \( z \in B \) such that

\[
\forall x \in S. x \sqsubseteq z, \text{ and } \forall b \in B. x \sqsubseteq b \supseteq z \sqsubseteq b
\]

For \( S \subseteq B \), \( S \) is a directed subset of \( B \) iff \( S \) is non-empty and every finite subset of \( S \) has a lub in \( S \). A complete partial order, abbreviated cpo, is a partial order \( (B, \sqsubseteq) \) with a minimum element, denoted \( \bot \) such that every directed subset has a lub in \( B \).
A common example of a domain is the powerset domain. Each element in the domain \( \mathcal{P}(A) \) is some subset of the set \( A \). The least element is the empty set, and the approximation ordering is the subset relation.

2.3.2 Domain Constructors

New domains can be constructed from existing domains. The domain constructions that are important in the denotational definitions presented here are products, sums, infinite sequences, lifted spaces, function spaces, and finite functions.

Product Spaces

A product domain contains elements that are tuples of a given length. The domain \( A \otimes B \) contains pairs of non-bottom elements, with an additional least element:

\[
A \otimes B = \{ \bot \} \cup \{(a, b) \mid a \in A - \{ \bot \} \land b \in B - \{ \bot \}\}
\]

The constructor \((\cdot, \cdot)\) creates an element of \( A \otimes B \) as follows:

\[
(a, b) = \begin{cases} 
(a, b) & \text{if } a \not= \bot \land b \not= \bot \\
\bot & \text{otherwise}
\end{cases}
\]

The expression \((e_1, e_2, \ldots, e_n)\) abbreviates the expression \((e_1, (e_2, \ldots, (e_{n-1}, e_n))\)).

The approximation relations for the component domains of a product space define the approximation relation for the product space as follows:

\[
\text{for } (a_1, b_1), (a_2, b_2) \in A \otimes B, (a_1, b_1) \sqsubseteq_{A \otimes B} (a_2, b_2) \iff a_1 \sqsubseteq_A a_2 \land b_1 \sqsubseteq_B b_2
\]

Lifted Spaces

The domain \( A_\bot \) is the domain \( A \) with a new \( \bot \):

\[
A_\bot = \{ \bot \} \cup \{(T, a) \mid a \in A\}
\]

This lifting operation distinguishes between the least element \( \bot_A \) from the original domain \( A \) and the new \( \bot \) such that \( \bot \sqsubseteq \bot_A \). Otherwise, the approximation relation for the lifted space is the approximation relation for the original domain \( A \).
Infinite Sequences

The infinite sequence domain \( A^+ \) defines a domain whose elements are potentially infinite sequences of elements from \( A \):
\[
A^+ = A \otimes (A^+)_\bot
\]

The expression \([e_1, e_2, \ldots]\) abbreviates the expression
\[
\langle e_1, (T, (e_2, (T, \ldots)\rangle)
\]

The function \( a_0 \circ [a_1, a_2, \ldots] \) appends the element \( a_0 \) on the front of the sequence as follows:
\[
a_0 \circ [a_1, a_2, \ldots] = \langle a_0, (T, [a_1, a_2, \ldots])\rangle = [a_0, a_1, a_2, \ldots]
\]

Sum Spaces

A sum domain contains tagged elements from the component domains. The domain \( A \oplus B \) contains non-bottom elements from domain \( A \) tagged with a \( T \), non-bottom elements from domain \( B \) tagged with an \( F \), and a distinct bottom element:
\[
A \oplus B = \{ (T, a) \mid a \in A - \{ \bot \} \} \cup \{ (F, b) \mid b \in B - \{ \bot \} \} \cup \{ \bot \}
\]

The approximation relations for the component domains of a sum space define the approximation relation for the sum space as follows:

for \( \langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle \in A \oplus B, \langle a_1, b_1 \rangle \sqsubseteq_{A \oplus B} \langle a_2, b_2 \rangle \) iff \( a_1 \sqsubseteq_A a_2 \land b_1 \sqsubseteq_B b_2 \)

Function Spaces

The elements of the domain \( A \rightarrow B \) are continuous functions with elements in domain \( A \) as the input to the function and domain \( B \) as the codomain of the function.

**Definition 2.9 (Monotonic and Continuous Functions)** A function \( f \in A \rightarrow B \) is monotonic if
\[
\forall a, a' \in A, a \sqsubseteq_A a' \supseteq f(a) \sqsubseteq_B f(a').
\]

A function \( f \in A \rightarrow B \) is continuous if it is monotonic and for all directed sets \( S \subseteq A \)
\[
\bigcup \{ f(s) \mid s \in S \} = f( \bigcup S)
\]
The domain $A \rightarrow B$ is the set of all finite functions from $A$ to $B$. We frequently treat elements of $A \rightarrow B$ as elements of $A \rightarrow B$.

### 2.3.3 Denotational Definitions

A denotational description of a language consists of three parts: a definition of the abstract syntax of the language, a specification of the relevant semantic domains, and the meaning function.

The semantic domains are defined in terms of primitive domains and the domain constructors described previously. For this dissertation, we make few assumptions about the primitive domain $val$ which is the domain containing the final answers for programs. First, we assume that the domain $val$ is flat, meaning that there are no infinite sets $v_i \subseteq val$ such that $v_0 \subseteq v_1 \subseteq v_2 \subseteq \ldots$. In addition, the domain $val$ must contain the domain $bool = \{\bot, \text{true}, \text{false}\}$.

The meaning functions are recursive functions on the syntax. Since the clauses in the meaning function specify the meaning in terms of the meaning of the parts of the statements and are continuous functions over domains, the meaning functions are well-defined and exist [47, 51].

A meaning function maps a program to its denotation. The denotations for programs in most programming languages are functions that map input stores to output stores, also referred to as store transformers. Definitions that require a continuation argument [53, 46] are called continuation-passing semantics. Definitions for languages that do not require continuations are referred to as direct semantic definitions.

In the following chapters, an algorithm to create $pdgs$ is derived from the denotational definition of the programming language by a process referred to as a staging transformation [33]. This process splits a denotational definition into a static component, the compiler, and a dynamic component, the interpreter. The compiler translates the program text into some natural representation for the information required by the interpreter so that it can the denotation of the program specified by the original denotational definition. This process provides a compiler and an interpreter for the language that maintain the meaning of the program with respect to the original denotational definition for the language.
2.4 Program Dependence

This section defines the different dependence relations discussed in the literature. Informally, a dependence exists between two statements when the source statement of the dependence has an effect on the computation at the target statement of the dependence. The dependence definitions assume the program has been translated into its control flow graph. Several of the dependence definitions rely on the concept of a reaching assignment [2].

**Definition 2.10 (Reaching Assignment)** An assignment to $x$ in node $n$ reaches node $m$ if there is a path $p$ from node $n$ to node $m$ with no other node containing an assignment to $x$.

A flow dependence [35] connects an assignment node for identifier $x$ to a node that uses this assignment for $x$.

**Definition 2.11 (Flow Dependence)** A node $m$ is flow dependent on node $n$, denoted $n \rightarrow_f m$, if node $n$ contains an assignment to an identifier $x$, the assignment to $x$ reaches $m$, and $m$ contains a reference to the identifier $x$.

Output dependences [35] sequence multiple assignments to the same identifier.

**Definition 2.12 (Output Dependence)** A node $m$ is output dependent on node $n$, denoted $n \rightarrow_o m$, if node $n$ and node $m$ both assign a value to the same identifier $x$ and the assignment to $x$ in $n$ reaches node $m$.

Output dependence is the same as a flow dependence if an assignment to an identifier is also considered a use of that identifier.

Def-order [31] dependence is a restriction of the transitive closure of the output dependence relation.

**Definition 2.13 (Def-Order Dependence)** A node $m$ is def-order dependent on node $n$, denoted $n \rightarrow_d m$, if node $n$ and node $m$ both assign a

---

5Kuck refers to this type of dependence as a true dependence.

6There is a symmetric concept to output dependence called input dependence [35] which sequences uses of the same identifier. Input dependences are not considered in this dissertation as they are not required to maintain the correctness of the program.
value to identifier $x$, there exists a node $n'$ such that $n'$ uses $x$ and the assignments for $x$ in $n$ and $m$ both reach $n'$, and there is a path from $n$ to $m$. The node $n'$ is called the witness node for the def-order dependence.

Def-order dependences apply only to assignment nodes that have a common use of the assigned identifier. Thus, sequencing among multiple assignments to an identifier that are not used in any expression is not specified by a def-order dependence. The following program illustrates the distinction between def-order and output dependence:

```plaintext
1  x := 1;
2  x := 2;
3  x := 3;
4  if p
5      then x := 4
6        else y := 1;
7  z := x
```

For this program, there is a def-order dependence from statement 3 to statement 5, but there are output dependences from statement 1 to statement 2, statement 2 to statement 3, and statement 3 to statement 5. The meaning of the program does not change if the order of statements 1 and 2 is reversed.

The next example illustrates the presence of a def-order dependence that does not correspond to an output dependence:

```plaintext
1  x := 1;
2  if p
   then
   3    x := 4
   4    x := 5
   6    else x := 2;
   7  z := x
```

In this example, there are output dependences from statement 1 to statement 3, statement 1 to statement 6, and statement 3 to statement 4, but the def-order dependences are from statement 1 to statement 4 and statement 1 to statement 6.

Anti-dependences [35] are storage dependences that ensure the value for an identifier is not over-written before it is used.
Definition 2.14 (Anti-Dependence) Node $m$ has an anti-dependence from node $n$, denoted $n \rightarrow_a m$, if node $m$ assigns a value to identifier $x$, node $n$ uses $x$, and there is a path $p$ from node $n$ to node $m$ with no assignment to $x$.

Anti-dependences only arise in situations where the values for identifiers reside in a central store and thus locations in the store are reused for unrelated computations. The following program illustrates this situation:

1. $x := 1$
2. $y := f(x)$
3. $x := 4$
4. $z := g(x)$

The assignments to $x$ in statements 1 and 3 are unrelated unless the values are recorded in a shared store. In that case, the assignment in statement 3 must wait until the retrieval of the value of $x$ in statement 2 has occurred, giving rise to the anti-dependence from statement 2 to statement 3. If there is no shared store, this dependence is not necessary. Renaming of identifiers [16, 19] also breaks this dependence by changing the program so that the identifier names are not re-used in this manner.

The control dependence relation [24, 20] characterizes the conditions under which a node executes.\footnote{To simplify the definition of control dependence, it is assumed that there is an edge added to the control flow graph of a program from the start node to the end node labeled $\mathbf{F}$, and that the edge from the start node to the first node of the program is labeled $\mathbf{T}$.}

Definition 2.15 (Control Dependence) A node $m$ is control dependent on node $n$ with label $b \in \{\mathbf{T}, \mathbf{F}\}$ if there exists a path $p$, whose first edge is labeled $b$, from $n$ to $m$ such that any node $q$ on path $p$ excluding $n$ and $m$ is post-dominated by $m$ and $n$ is not strictly post-dominated by $m$.

Intuitively, $m$ is control dependent on $n$ if there is a path from $n$ to the end node that includes $m$ and at least one path from $n$ to the end node that does not include $m$. The restriction on the other nodes in the path finds the immediate control dependence predecessor. By the definition of the strict post-dominator relation, a node does not strictly post-dominate itself. Therefore, a node can be control dependent
on itself. Predicate nodes for loops are control dependent on themselves. In structured programs, the control dependence relation reflects the nesting structure of the program, with the additional self-dependences on the loop predicate nodes.

2.5 Program Dependence Graphs

The program dependence graphs, \( pdg \), as initially defined by Ferrante et al. [24], combines the information in the control flow graph with the dependence information. The original claim for the \( pdg \) is that this representation contained all of the sequencing constraints required to maintain the meaning of the program, but no support of this claim was presented. This work finally satisfies this claim.

**Definition 2.16 (Program Dependence Graphs)** Let \( Exp \) be a set of expressions and \( Id \) a set of identifiers, deliberately unspecified. A \( pdg \) \( G = (N, E) \) is a potentially cyclic, finite directed graph where

\[
N = Exp \cup (Id \times Exp) \cup \{\text{start}\}
\]

and

\[
E \subseteq N \times N
\]

The node set can be partitioned into three disjoint sets: predicate nodes, assignment nodes, and the designated \textbf{start} node. The edge set can be partitioned into eight sets: loop-carried and loop-independent flow dependence, output dependence, anti-dependence, and control dependence edges. Control dependence edges are labeled either \( T \) or \( F \). Control dependence edges must have either the \textbf{start} node or a predicate node as the source node; the other edges must have an assignment node as the source node. The start node has no incoming edges.

The nodes in the \( pdg \) are the same nodes that exist in the control flow graph, except that there is no \textbf{end} node in the \( pdg \). Cycles in a \( pdg \) arise from loops in the program. All types of edges can cause cycles in the \( pdg \).

The \( pdg \) defined above is the basis for several dependence-based program representations. Other formulations are described in the conclusion. None of the presentations, however, have provided a semantic characterization of \( pdg \) or dependence. This dissertation uses denotational semantics to explore the nature of program dependence and to provide a mathematical foundation for those program manipulation
tools that use the \textit{pdg} or other dependence-based representations. This process results in a structure that is quite similar to the \textit{pdg} as defined above, is semantically sound, can be used for the same purposes as those proposed for the \textit{pdg}, and is superior in some instances to the traditional \textit{pdg} for optimization purposes. The differences between the \textit{pdg} derived here and the traditional \textit{pdg} arise from two sources: changes necessary to specify a correct semantics for the \textit{pdg}, and changes necessary to preserve compositionality in the semantics for the \textit{pdg}. 
Chapter 3

Semantics of Structured Program Dependence

Initially, the study of the semantics of program dependence focused on a simple structured language. This chapter reviews this study and introduces the techniques used to describe the language features added in the later chapters. The first section of this chapter describes the language itself, presenting the syntax and the denotational semantics. Section 3.2 presents a characterization of dependence in terms of a denotational semantics for \textit{pdt}s, a structure which closely resembles the \textit{pdg}. The operational semantics for the \textit{pdt}s derived in Section 3.2 is the subject of Section 3.3.

3.1 Semantics for Language $\mathcal{W}$

The structured language $\mathcal{W}$ supports assignment statements, if statements, while statements, and statement composition:

\[
\begin{align*}
  i, x, x; & \in \text{ide} \\
  e, p & \in \text{exp} \\
  s & ::= \text{if } e \text{ } s \text{ while } e \text{ } s \mid x ::= e \mid s; s \\
  P & ::= s; \text{end}
\end{align*}
\]

where $s$ ranges over statements, $x$ ranges over the set of identifiers and $e$ ranges over expressions. The expression language is left unspecified although we make several basic assumptions about the language and its semantics. First, we assume there are boolean expressions. We also assume that expressions may refer to identifiers, and that the value of an expression is a deterministic function of its free identifiers: $\mathcal{E}[e]_\sigma$ denotes the evaluation of expression $e$ with the values for the free identifiers in $e$ found in the store $\sigma$. Expression evaluation is assumed to satisfy the Substitution Lemma. Thus, the following equality holds for expression $e$, identifier $x$, value $v$ and store $\sigma$:

\[
\mathcal{E}[e]_\sigma[x \leftarrow \mathcal{E}[v]_\sigma] = \mathcal{E}[e[x \leftarrow v]]_\sigma
\]

Finally, we assume that expressions are pure, with no laziness in the expression language, and that the expression evaluation function is strict. The domain of values is
also unspecified, except that we assume the domain is flat, and that \( \bot \) denotes the least element in the domain.

Two functions \( R \) and \( A \) provide information on the uses and assignments of identifiers in statements and expressions. The function \( R \) maps expressions to the finite set of identifiers that occur in that expression; the function \( A \) is a predicate that determines if an identifier is assigned in a statement. Figure 3.1 contains the definitions of the function \( A \). The function \( R \) is unspecified as it depends on the particular expression language chosen.

### 3.1.1 Denotational Semantics for Language \( W \)

Figure 3.2 presents the denotational semantics for language \( W \). The meaning function assigns the standard sequential interpretation to the program. A program’s meaning is a store transformer, where a store is a finite function mapping identifiers to values. The clause for assignment statements, \( A \), specifies the meaning of an assignment statement as the functional update of the store. The store update function, \( \delta_s \) is strict in all three arguments; this update is defined in Figure 3.3.

The meaning of if statements, given in clause \( I \), is the meaning of either the true branch or the false branch, depending on the value of the predicate expression \( p \). The clause for statement composition, \( C \), composes the meanings of the component statements by using the resulting store from the first statement as the store argument for the second statement. Finally, the clause for the while statement, \( W \), composes the effects of repeated evaluation of the while body until the controlling predicate evaluates to false. The function fix is the least fixed point function.

\[
\begin{align*}
A &: stmt \to ide \to bool \\
A[x := e] &= \lambda i . i \uparrow = x \to T, F \\
A[if \ p \ s_t \ s_f] &= \lambda i . A[s_t]i \lor A[s_f]i \\
A[s_1; s_2] &= \lambda i . A[s_1]i \lor A[s_2]i \\
A[while \ e \ s] &= \lambda i . A[s]i \\
A[end] &= \lambda i . F
\end{align*}
\]

*Figure 3.1* Definition for \( A \)
Semantic Domains:

\[ v \in \text{val} \quad \text{(abstract)} \]
\[ x \in \text{ide} \]
\[ e, p \in \text{exp} \quad \text{(abstract)} \]
\[ \sigma \in \text{store} = \text{ide} \rightarrow \text{val} \]

Semantics:

\[ \mathcal{E} : \text{exp} \rightarrow \text{store} \rightarrow \text{val} \quad \text{(abstract)} \]
\[ \mathcal{W} : \text{stmt} \rightarrow \text{store} \rightarrow \text{store} \]

\text{A} : \quad \mathcal{W}[x := e] = \lambda \sigma. \delta_S(\sigma, x, \mathcal{E}[e] \sigma) \\
\text{I} : \quad \mathcal{W}[\text{if } p\ s_i\ s_f] = \lambda \sigma. \mathcal{E}[p] \sigma \rightarrow \mathcal{W}[s_i] \sigma, \mathcal{W}[s_f] \sigma \\
\text{C} : \quad \mathcal{W}[s_1; s_2] = \lambda \sigma. \mathcal{W}[s_2](\mathcal{W}[s_1] \sigma) \\
\text{W} : \quad \mathcal{W}[\text{while } p\ s] = \text{fix}(\lambda w. \lambda \sigma. \mathcal{E}[p] \sigma \rightarrow w(\mathcal{W}[s] \sigma), \sigma) \\
\text{F} : \quad \mathcal{W}[s; \text{end}] = \lambda \sigma. \mathcal{W}[s] \sigma

**Figure 3.2** Denotational Definition of Language \( \mathcal{W} \)

\[
\delta_S(\sigma, x, v)(y) = \begin{cases} 
\bot & \text{if } \sigma = \bot \lor x = \bot \lor v = \bot \\
\sigma(y) & \text{if } y \neq x \\
v & \text{if } y = x 
\end{cases}
\]

for all \( \sigma, x, y, \) and \( v \).

**Figure 3.3** Definition of the Strict Store Update Function
3.2 Dependence Trees for Language $\mathcal{W}$

This section reviews the semantic characterization of pdts for structured programs developed by Cartwright and Felleisen [15, 14]. This derivation serves as the basis for the denotational derivations presented in subsequent chapters.

3.2.1 Generalized Semantics for Language $\mathcal{W}$

The analysis begins with the traditional denotational definition of language $\mathcal{W}$, presented in Figure 3.2 in the previous section. This meaning function, using the traditional strict store update function, specifies a strict, sequential interpretation of a program. Strictness implies that if any input is the value $\bot$ at any stage in the evaluation, the result must be $\bot$ as well. In terms of the semantic function, since the store update is strict, any statement that updates the store with a diverging value for some identifier causes the entire program to diverge, whether or not that value for that identifier is ever accessed. Intuitively then, each statement of a program depends on every statement that may precede it on some program execution. This sequential semantics and the corresponding dependence relationship is inconvenient for both traditional program optimization and parallelization. Since the programmer requires the semantics of an optimized program to reason about the behavior of the program, a semantics supporting the optimization process is appropriate for the study of dependence. To relax the sequentiality of the semantics to support these optimizations, the strictness of the update function is modified. There are two non-strict generalizations of the update function that are relevant to this analysis.

The first generalization, referred to as the lazy update function, is non-strict in both the store and value argument positions. Specifically, the lazy update function is defined as follows:

$$
\delta_2(\sigma, x, v)(y) = \begin{cases} 
\bot & \text{if } x = \bot \\
\sigma(y) & \text{if } x \neq \bot \land y \neq x \\
v & \text{if } y \neq \bot \land y = x 
\end{cases}
$$

With the lazy update function, the result store is defined if $x$ is defined and either $v$ or $\sigma$ is defined. Conceptually, the computation of a value is suspended until the value is retrieved from the store. Thus, if a diverging value is assigned to an identifier, but the identifier is redefined to a converging value before the value is requested, the diverging value does not affect the result.
The second generalization, referred to as the lackadaisical update, is only non-strict in the value argument. Specifically, the update for an identifier $x$ requires the store to contain a converging value for $x$. Thus, the $\perp$ value is a "sticky" value; once an identifier is assigned a diverging value, the value of that identifier is $\perp$, regardless of subsequent assignments. The lackadaisical update function, $\delta_K$ is defined as follows:

$$
\delta_K(\sigma, x, v)(y) = \begin{cases} 
\perp & \text{if } x = \perp \\
\sigma(y) & \text{if } x \neq \perp \land y \neq x \\
v & \text{if } y \neq \perp \land y = x \land \sigma(x) \neq \perp \\
\perp & \text{if } y \neq \perp \land y = x \land \sigma(x) = \perp 
\end{cases}
$$

for all $\sigma, x, y, v$. This update function separates the effects of assignments to different identifiers. However, all computations for an identifier $x$ may affect the final value of $x$, since any diverging value assigned to $x$ forces the value for $x$ to be $\perp$.

Clearly, the lazy update function dominates the other two update functions, with the lackadaisical update function dominating the strict update function:

$$
\delta_S \subseteq \delta_K \subseteq \delta_Z
$$

By abstracting the meaning function $\mathcal{W}$ with respect to the store update function, yielding $\mathcal{W}_\sigma$, three meaning functions can be found as instantiations of the generalized function:

$$
\begin{align*}
\mathcal{W}_S &= \mathcal{W}_\delta(\delta_S) \\
\mathcal{W}_K &= \mathcal{W}_\delta(\delta_K) \\
\mathcal{W}_Z &= \mathcal{W}_\delta(\delta_Z)
\end{align*}
$$

Given the relationship among the update functions and the monotonicity of $\mathcal{W}_\delta$, the following relationship holds among the meaning functions:

$$
\mathcal{W}_S \subseteq \mathcal{W}_K \subseteq \mathcal{W}_Z.
$$

The significance of this relationship is illustrated by the following example from Cartwright and Felleisen [15]. The program

$$
Q \triangleq y := 1; x := \perp; x := 2; \textbf{end}
$$

has different meanings under the three meaning functions:

$$
\begin{align*}
\mathcal{W}_S[Q] &= \lambda \sigma. \perp_{\text{store}} \\
\mathcal{W}_K[Q] &= \lambda \sigma. \sigma(y) \triangleq \perp \rightarrow \perp, \{(y, 1)\} \\
\mathcal{W}_Z[Q] &= \lambda \sigma. \{(x, 2), (y, 1)\}
\end{align*}
$$
The lazy semantics ignores all computations that are not required to compute the final values of the identifiers while the lackadaisical semantics ignores undefined values for identifiers that are not demanded or required to compute the value for the demanded identifiers. These perspectives result in different assumptions that a compiler can make in code optimization and parallelization. For example, the lazy semantics separates the meanings of two different assignments to the same identifier unless the first value is required to compute the second value. Under the lackadaisical semantics, the first assignment must complete before the second assignment can begin, even if there is no other relationship among the statements. These perspectives also give rise to different views of data dependence and thus different dependence relations. The next section explores these dependence relations.

3.2.2 Data Dependence and Demand Semantics

Data dependence analysis attempts to determine which subcomputations are required to compute the value of an expression, and where these computed values are used in subsequent computations. The above semantic functions are based on the store model of computation, associating an identifier with its final value in the result store. A value flow model of computation, however, is better suited to answering the data dependence questions, since this model associates an identifier with the computations required to obtain the value of that identifier.

Simply applying the result store to the identifier in question is insufficient, because this still treats the preceding text of the program as a store transformer. Instead, we transform the meaning function $W_s$ of type

$$W_s : update \to stmt \to (ide \to val) \to (ide \to val)$$

into a new meaning function $V$ of type

$$V : update \to stmt \to ide \to (ide \to val) \to val$$

to obtain a value flow model from the store model. The new semantic function associates a value with a demanded identifier and a store as opposed to associating an output store with an input store. This property holds for intermediate computations as well as for the final values under this transformation. Figure 3.4 presents the altered semantics, called the demand semantics since the evaluation is driven by demands for the values of identifiers. The statement composition clause, $C_V$, in conjunction with
Semantic Domains:

\[ \begin{align*}
  v & \in val \quad \text{(abstract)} \\
  i, x & \in ide \\
  e, p & \in exp \quad \text{(abstract)} \\
  \sigma & \in store = ide \rightarrow val \\
  \delta & \in update = store \rightarrow ide \rightarrow val \rightarrow store
\end{align*} \]

Semantics:

\[ \begin{align*}
  \mathcal{E} & : \exp \rightarrow store \rightarrow val \\
  \mathcal{V} & : \update \rightarrow \stmt \rightarrow ide \rightarrow val
\end{align*} \]

\[ \begin{align*}
  A_V : \forall \delta[x := e] & = \lambda i.\sigma.\delta(\sigma, x, E[e]\sigma)i \\
  I_V : \forall \delta[\text{if } p \ s_1 \ s_2] & = \lambda i.\sigma.\mathcal{E}[p]\sigma \rightarrow \forall \delta[s_1]i\sigma, \forall \delta[s_2]i\sigma \\
  C_V : \forall \delta[s_1; s_2] & = \lambda i.\sigma.\forall \delta[s_1]i(\lambda j.\forall \delta[s_1]j\sigma) \\
  W_V : \forall \delta[\text{while } p \ s] & = \text{fix}(\lambda w.\lambda i.\sigma.\mathcal{E}[p]\sigma \rightarrow w(\lambda j.\forall \delta[s]j\sigma), \sigma i) \\
  F_V : \forall \delta[s; \text{end}] & = \lambda i.\sigma.\forall \delta[s]i\sigma
\end{align*} \]

\[ \text{Figure 3.4 Demand Semantics for Language } \mathcal{W} \]

the store accesses performed by the expression evaluator, shows how demands for values of identifiers propagate.

The connection between demand propagation, data dependence, and the different store update functions manifests itself in the assignment clauses. In the assignment clause with the strict update function, \( A^S_V \), evaluation of the expression occurs regardless of the specific demand. Thus, every statement depends on the successful completion of all previously evaluated statements, and the demand propagation has not altered the dependence structure. In the assignment clause for the lackadaisical update function, \( A^K_V \),

\[ \forall_i \llbracket x := e \rrbracket = \lambda i.\sigma.i \overset{?}{=} x \rightarrow (\sigma i \overset{?}{=} \bot \rightarrow \bot, \mathcal{E}[e]\sigma), \sigma i \]

the evaluation of the expression is ignored if the demand does not match the identifier in the assignment statement. However, since the value for the demanded identifier in the store is tested, all assignments for that identifier must successfully complete. Thus, an assignment for an identifier \( x \) depends on all previously evaluated assignments to \( x \). This dependence captures the notion of \textit{output} dependence discussed earlier.
The lazy update assignment clause, $A_{\nu}$,

$$V_{\nu}[x := e] = \lambda i\sigma.i \equiv x \rightarrow E[e]\sigma,\sigma i$$

ignores the computation of the expression $e$ if the demanded identifier is not the identifier in the assignment statement. The previous value for the identifier is not requested from the store.

The correspondence of the lazy update function to the notion of def-order dependence is not as direct as in the case of output dependence and the lackadaisical update function. The lazy semantics dominates the traditional, intuitive semantics attached to a def-order dependence since the demand propagation stops when the more recent assignment is found to apply. In structured languages, def-order dependences arise in two situations. First, if an identifier is assigned on one branch of an if statement but not the other, a def-order dependence exists between the assignment(s) reaching the if statement and the assignment within the body of the if. Similarly, if an identifier is assigned within the body of a while loop, a def-order dependence exists from the assignment(s) for that identifier that reach the while statement to the assignment in the while body. In both cases, the dependence only exists if there is a demand for the identifier that is reached by both these assignments. The def-order dependence is necessary in both cases to ensure that the expression evaluation uses the correct value. While all of the semantic functions ensure that only one value arrives at the use, the lazy function does not require evaluation of the earlier expression unless the branch with the more recent assignment is not evaluated. For example, the if clause,

$$V_{\delta}[if \ p \ s_t \ s_f] = \lambda i\sigma.E[p]\sigma \rightarrow V_{\delta}[s_t]\sigma, V_{\delta}[s_f]\sigma$$

gates the flow of the value from before the if statement with the if predicate. Since the lazy assignment clause does not demand the previous value for the identifier as does the lackadaisical clause, demand propagates to the assignment before the if only if the demand is not satisfied in the body of the if statement. Since the traditional def-order semantics is data-driven, the assignment before the if statement is evaluated in either case, and thus, the program diverges if the earlier expression diverges, regardless of the value of the predicate controlling the conditional definition of the identifier. A similar analysis applies to the while case.

With these changes to the semantics, the computations required to compute the value of an identifier are identifiable using the semantic function. Thus, the demand
semantics specifies the data dependence relation in a program. The semantic function $\mathcal{V}$ satisfies the equation

$$(\mathcal{W} \delta[p] i) \sigma = \mathcal{V} \delta[p] i \sigma$$

for all $i, p, \delta, \sigma$. Thus, for the same store update function, the change in processing to the demand perspective has not changed meaning of the program.

### 3.2.3 Control Dependence

The next step in the derivation develops the denotational equivalent of the control dependence relation. In structured programs, the control dependence relation reflects the nesting structure of the program. A control parameter is added to the semantic function to encapsulate the required control information. Figure 3.5 contains the revised clauses.

Several changes are required to accommodate the additional control parameter, $\kappa$. First, a more mathematical perspective is used in the interpretation of the if statement. The if clause in the $\mathcal{V}$ meaning function selected either the true or the false branch of the if statement based on the value of the predicate expression. This same behavior is obtained by attempting to evaluate both branches in parallel, returning the value $\bot$ for the branch that does not apply based on the value of the predicate expression, and then merging the values from the two branches. The function used for this merge operation is the least upper bound operation, $\sqcup$ over the domain $val^T$. The role of the value $T$ and its meaning is described below. Thus, the if clause for the control stage of the derivation,

$$\mathcal{C} \delta[[\text{if } p s_t s_f]] = \lambda i \kappa \sigma. \neg (\mathcal{A}(s_t, i) \lor \mathcal{A}(s_f, i)) \rightarrow \sigma i,$$

let $\kappa^+ = \kappa \land \mathcal{E}[p] \sigma, \kappa^- = \kappa \land \neg \mathcal{E}[p] \sigma$ in

$$(\neg \mathcal{A}(s_t, i) \rightarrow (\kappa^+ \rightarrow \sigma i, \bot), \mathcal{C} \delta[s_t] i \kappa^+ \sigma)$$

$\sqcup$

$$(\neg \mathcal{A}(s_f, i) \rightarrow (\kappa^- \rightarrow \sigma i, \bot), \mathcal{C} \delta[s_f] i \kappa^- \sigma)$$

uses the least upper bound operation to combine the meanings of the true and the false branch, and combines the predicate value with the incoming control value $\kappa$ as the control value for the branches of the if statement.

---

8The symbol $\sqcup$ in the domain equation $val \sqcup T$ does not refer to the union operation. This notation instead signifies the domain where the value $T$ is added to the domain $val$ and all elements of $val$ approximate $T$. 
Semantic Domains:

\[
\begin{align*}
\nu & \in \text{val}^T = \text{val} \cup \{\top\} \\
i, x & \in \text{ide} \\
e, p & \in \text{exp} \\
k & \in \text{bool}^T \\
\sigma & \in \text{store}^T = \text{ide} \rightarrow \text{val}^T \\
\delta & \in \text{update} = \text{store}^T \rightarrow \text{ide} \rightarrow \text{val}^T \rightarrow \text{store}^T
\end{align*}
\]

Semantics:

\[
\begin{align*}
\mathcal{E} : \text{exp} \rightarrow \text{store}^T \rightarrow \text{val}^T & \quad \text{(abstract)} \\
\mathcal{C} : \text{update} \rightarrow \text{stmt} \rightarrow \text{ide} \rightarrow \text{bool}^T \rightarrow \text{store}^T \rightarrow \text{val}^T
\end{align*}
\]

\[
\begin{align*}
\text{A}_\mathcal{C} : \quad & C\delta[x := e] = \lambda \iota \kappa.\sigma.\delta(\sigma, x, \mathcal{E}[e](\sigma)_i, \bot) \\
\text{I}_\mathcal{C} : \quad & C\delta[\text{if } p \ s_i \ s_f] = \lambda \iota \kappa.\sigma.\neg(\mathcal{A}(s_i, i) \lor \mathcal{A}(s_f, i)) \rightarrow \sigma_i, \\
& \quad \text{let } \kappa^+ = \kappa \land \mathcal{E}[p](\sigma), \kappa^- = \kappa \land \neg\mathcal{E}[p](\sigma) \text{ in} \\
& \quad (\neg\mathcal{A}(s_i, i) \rightarrow (\kappa^+ \rightarrow \sigma_i, \bot), C\delta[s_i]i\kappa^+\sigma) \\
& \quad \cup \\
& \quad (\neg\mathcal{A}(s_f, i) \rightarrow (\kappa^- \rightarrow \sigma_i, \bot), C\delta[s_f]i\kappa^-\sigma) \\
\text{C}_\mathcal{C} : \quad & C\delta[s_1; s_2] = \lambda \iota \kappa.\sigma.C\delta[s_2]i\kappa(\lambda j.C\delta[s_1]j\kappa\sigma) \\
\text{W}_\mathcal{C} : \quad & C\delta[\text{while } p \ s] = \text{fix}(\lambda \lambda w.\lambda \iota \kappa.\sigma.\neg\mathcal{A}(s, i) \rightarrow \sigma_i, \\
& \quad \text{let } \kappa^+ = \kappa \land \mathcal{E}[p](\sigma), \kappa^- = \kappa \land \neg\mathcal{E}[p](\sigma) \text{ in} \\
& \quad (\text{win}+(\lambda j.C\delta[s]j\kappa^+\sigma)) \cup (\kappa^- \rightarrow \sigma_i, \bot)) \\
\text{F}_\mathcal{C} : \quad & C\delta[s; \text{end}] = \lambda \iota \kappa.\sigma.C\delta[s]i\kappa\sigma
\end{align*}
\]

\textbf{Figure 3.5} Control Semantics for Language \(\mathcal{W}\)
The if clause, $I_C$, also includes a test to determine if the demanded identifier is assigned within the body of the if statement. To avoid dependences on predicate expressions that do not affect the value for the demanded identifier, a syntactic check of the body of the if statement is made, using the predicate $A$. If the predicate $A(s, x)$ is true, the identifier $x$ is assigned in the statement $s$. If the demanded identifier is not assigned in either the true or the false branch of an if statement, the demand propagates directly to the store $\sigma$, bypassing the entire if statement, including the evaluation of the predicate expression. This change causes a lifting of the meaning of programs since the presence of a diverging yet irrelevant predicate expression would cause the semantic function $V$ to diverge whereas that same program would converge under the semantic function $C$.

One other significant change in the if clause is the bypassing of demand propagation through a branch that does not contain a definition of the demanded identifier. This change explicates the def-order dependence. If the demanded identifier is not defined on a branch in the if statement, the demand propagates directly to the store $\sigma$.

The use of the least upper bound function requires a change in the domains for the meaning function. Specifically, a top value, $\top$, must be added to the value domain $val$ to ensure that the meaning function is defined over the whole domain. The value $\top$ is defined to be above all other values of the domain $val$; thus, the domain $val^\top$ is a lattice and the least upper bound function is total and continuous. The other domains built using the domain $val$ are altered to use the $val^\top$ domain. No program in language $\mathcal{W}$ denotes the value $\top$.

**Theorem 3.1** For statement $s \in \mathcal{W}$, identifier $i$, boolean value $\kappa$ and store $\sigma$ such that $\forall j. \sigma j \neq \top$

$$C[s][i \kappa \sigma] \neq \top$$

**Proof** The proof proceeds by induction on the structure of the statement $s$.

**Case** $x := e$ Since the expression evaluation function does not return the value $\top$ unless $\sigma j = \top$ for some identifier $j$, this case follows immediately.
Case if $p\ s_i\ s_f$ If $i$ is not assigned in the body of the if statement, the result holds based on the assumption about the store $\sigma$. Otherwise, the induction hypothesis gives us that

$$C[s_i]i\kappa\sigma \neq T$$

and

$$C[s_f]i\kappa\sigma \neq T$$

There are several cases to consider. If $\kappa = \bot$, the result is obvious. If $\kappa = \text{false}$, then $\kappa^+$ and $\kappa^-$ are both false. The result follows directly from Lemma 3.1. If $\kappa = \text{true}$, at most one of $\kappa^+$ or $\kappa^-$ is true. Thus, the result holds using the induction hypothesis, Lemma 3.1 and the assumption on the store $\sigma$.

Case $s_1; s_2$ Using the induction hypothesis, the store $\lambda j.C[s]j\kappa\sigma$ fulfills the assumptions of the theorem. Thus, the induction hypothesis holds for statement $s_2$ and the result is shown.

Case while $p\ s_b$ If $\kappa = \bot$, the result is obvious. If $\kappa = \text{false}$, all of the $\kappa^+$'s and $\kappa^-$'s formed by the while clause are also false or $\bot$ and the result follows immediately. If $\kappa = \text{true}$, at most one $\kappa^-$ can be true by the construction of $\kappaappa^+$ and $\kappa^-$. By the induction hypothesis and the assumption on the initial store, only one component of the lub is not $\bot$ and this value is not $T$. Thus, the result holds.

$\square$

Lemma 3.1 Let $s \in \mathcal{W}$ be a statement, $i$ an identifier, and $\sigma$ a store. Then,

1. If $\mathcal{A}(s,i) = \text{true}$ and $\kappa = \text{false}$ then

$$C[s]i\kappa\sigma = \bot$$

2. If $\mathcal{A}(s,i) = \text{false}$, then

$$C[s]i\kappa\sigma = \sigma i$$

Proof The proof proceeds by induction on the structure of the statement $s$. 
Case $x := e$ For $\mathcal{A}(s, i)$ to be true, $i = x$ must hold. Part 1 of the theorem follows directly. Otherwise, if $\mathcal{A}(s, i)$ is false, $i \neq x$ must hold. Part 2 follows directly from the definition of $\mathcal{C}$.

Case if $p.s_i, s_f$ For Part 2, if $\mathcal{A}(s, i)$ is false, then both $\mathcal{A}(s_i, i)$ and $\mathcal{A}(s_f, i)$ are false, by the definition of $\mathcal{A}$, and Part 2 holds. For Part 1, since $\kappa$ is false, $\kappa^+$ and $\kappa^-$ are both either false or $\bot$. The result follows directly if either value is $\bot$. Otherwise, the result follows using the induction hypothesis and the definition of $\bigvee$ and $\mathcal{C}$.

Case $s_1; s_2$ If $\mathcal{A}(s, i)$ is false, then both if $\mathcal{A}(s_1, i)$ and $\mathcal{A}(s_1, i)$ are false. Part 2 follows using the induction hypothesis for both $s_1$ and $s_2$. For Part 1, if $\mathcal{A}(s, i)$ is true, either $\mathcal{A}(s_1, i)$ is true, or $\mathcal{A}(s_2, i)$ is true or both. If $\mathcal{A}(s_2, i)$ is true, Part 1 of the theorem holds using the induction hypothesis for $s_2$. Otherwise, $\mathcal{A}(s_1, i)$ must be true and $\mathcal{A}(s_2, i)$ false. By the induction hypothesis, Part 2, the denotation of $s_2$ is

$$(\lambda j.\mathcal{C}[s_1][j\kappa]\sigma)i$$

By the induction hypothesis, Part 1, however, this denotes $\bot$ since $\mathcal{A}(s_1, i)$ is true.

Case while $p.s_i$ Part 2 is obvious. For Part 1, if $\kappa$ is false, all the $\kappa^-$ values are either false or $\bot$. If the values are false, all components of the lub are bottom, and the lemma holds. If any $\kappa^-$ value is $\bot$, the result is immediate.

\[\square\]

The while clause,

$$\mathcal{C}b[\text{while } p \text{ s}] = \text{fix}(\lambda w.\lambda i.\kappa\sigma.\neg\mathcal{A}(s, i) \rightarrow \sigma i, \text{ let } \kappa^+ = \kappa \land \mathcal{E}[p] \sigma, \kappa^- = \kappa \land \neg\mathcal{E}[p] \sigma \text{ in } (w i \kappa^+ (\lambda j.\mathcal{C}[s][j\kappa^+\sigma]) \bigvee (\kappa^- \rightarrow \sigma i, \bot))$$

contains changes similar to those made in the if clause. The same lifting occurs with the check for a definition of the demanded identifier in the body of the while statement. The result of the while clause is the least upper bound over the result
for each of the infinite possible iterations of the loop. At most one of the values will not be \( \bot \) so the result will not be \( T \).

The statement composition clause, \( C_C \), passes the control parameter along without modification. This clause demonstrates that statements on the same syntactic level have the same control dependences.

The assignment clauses use the control parameter to determine if the value \( \bot \) should be returned or if the expression should be evaluated. The general clause is straightforward. The lazy and lackadaisical clauses incorporate the control parameter in an unexpected way. For example, the lazy assignment clause,

\[
C_Z[x := e] = \lambda i \kappa \sigma. i \xrightarrow{z} x \to (\kappa \to E[e] \sigma, \bot), \sigma i
\]

first checks to see if the assigned identifier matches the demanded identifier. If not, the demand is propagated to the store, regardless of the value of the control parameter. Since the if and while clauses do not propagate a demand unless the identifier is assigned in the component statement, this formulation of the assignment clause does not propagate a non-bottom value when the control parameter \( \kappa \) is \texttt{false}. Theorem 3.1 proves this property. The same observation applies to the lackadaisical assignment clause.

The new semantic function \( C \) satisfies the following invariant with respect to the function \( V \):

\[
V \delta[p] i \sigma \subseteq C \delta[p] i T \sigma
\]

This approximation can be made an equality by removing the syntactic checks preventing the evaluation of irrelevant predicate expressions.

The semantic function \( C \) identifies all the essential control and data dependences for computing the value of any given identifier. The following example program illustrates this process, where we assume the lazy update function and that the demanded identifier is \( z \):

\[
\begin{align*}
x &:= 42; \\
q &:= h(x); \\
\textbf{if } p(x) \\
& \quad \textbf{then } y := f(x) \\
& \quad \textbf{else } q_1 := 6; \\
z &:= g(x, y);
\end{align*}
\]
The demand for \( z \) propagates first to the final assignment statement. The lazy assignment clause determines that this update indeed satisfies the demand and begins the evaluation of the expression \( q(x, y) \). This expression evaluation propagates demands for both \( x \) and \( y \). The demand for \( x \) first passes to the if statement. Since \( x \) is not assigned a value within the body of the if, the demand propagates through the store to the assignment to \( q \). This assignment does not satisfy the demand, so the demand passes to the assignment for \( x \). Since this assignment does satisfy the demand, the value of its expression is the result. The connection between the demand for the value of \( x \) and the assignment providing that value is established; all other statements are bypassed until the appropriate statement is found. This connection is a data dependence. For all of these statements, the control parameter has remain unchanged at the initial value of \( T \).

In the case of the demand for \( y \), since \( y \) is assigned a value within the body of the if statement, the demand is propagated inside the if statement. The if clause computes control parameters for both the true and the false branches and then determines to which branch or branches to send the demand. Since an assignment for \( y \) appears on the true branch, the demand propagates to the statements in the then clause of the if statement with a control parameter formed from both the initial control parameter and the result of the evaluation of the predicate expression \( p(x) \). This step forms the connection between the predicate expression and those statements which rely on this predicate to determine if the statements are eligible for execution, which is the condition defining a control dependence.

3.2.4 Program Dependence Trees

The next step in the derivation prepares the semantic function for the staging that splits the function into a static analysis component and a dynamic evaluation component. To prepare for this split, all parameters that depend on the value of the initial store \( \sigma \) are identified and abstracted with respect to the initial store. The control parameter and the intermediate stores built in the statement composition clause and the while clause depend on the initial store. For the control parameter, by abstracting with respect to the store and replacing the \( \text{bool}^T \) parameter with a \( \text{store}^T \to \text{bool}^T \) parameter, the control parameter constructed in each clause contains all the information necessary to compute the boolean value once the store is available. However, an intermediate store parameter of type \( \text{store}^T \to \text{store}^T \) loses
the connection between the value of an identifier and the evaluation of individual statements. Thus, the identifier parameter is moved from the output store, as in the data dependence step earlier, to give a function of the type $ide \rightarrow store^T \rightarrow val^T$. The semantic function $C'$ now has the type

$$
\begin{align*}
stmt & \rightarrow ide \rightarrow (store^T \rightarrow bool^T) \\
& \quad \rightarrow (ide \rightarrow store^T \rightarrow val^T) \\
& \quad \rightarrow store^T \rightarrow val^T
\end{align*}
$$

All occurrences of a store parameter $\sigma$ in the function $C'$ refer to the initial store. The new semantic function $C'$ satisfies the following invariant:

$$
C'd[p][i](\lambda \sigma. T)(\lambda i \sigma. i) \sigma_0 = Cd[p][i]T \sigma_0
$$

Figure 3.6 contains the clauses for the function $C'$. This definition shows which parts of the function require the initial store and which decisions are independent of the initial store. For example in the lazy assignment clause,

$$
C'_Z[x := e] = \lambda i \kappa \gamma. i \Downarrow x \rightarrow (\lambda \sigma. \kappa \sigma \rightarrow E[e](\lambda j. \gamma j \sigma), \bot), \lambda \sigma. \gamma i \sigma
$$

the test to determine if this assignment statement applies to the demand is independent of the initial store. The computation of the control parameter, however, does require the initial store, as does the extraction of the result from the intermediate store, $\gamma$, if this assignment does not satisfy the demand.

The final step of the derivation is the separation of the function $C'$ into a static analysis function $G$ and a dynamic evaluation function $P$. The function $G$ builds program dependence trees, pdts, that are interpreted by the function $P$ once the initial store is available. Figure 3.7 presents the domain definitions for the semantic functions while Figure 3.8 contains the functions $G$ and $P$. The functions $G$ and $P$ must satisfy the following invariant:

$$
P(Gd[P][i](true_c) \gamma_0) \sigma_0 = C'd[P][i](\lambda \sigma. T)(\lambda i \sigma. i) \sigma_0 \quad (3.1)
$$

The required value for $\gamma_0$ is specified later in this section.

The following description of the lazy assignment clause explains the separation. The lazy assignment clause from the function $C'$,

$$
C'_Z[x := e] = \lambda i \kappa \gamma. i \Downarrow x \rightarrow (\lambda \sigma. \kappa \sigma \rightarrow E[e](\lambda j. \gamma j \sigma), \bot), \lambda \sigma. \gamma i \sigma
$$
Semantic Domains:

\[ v \in val^T = val \cup \{T\} \quad \text{(abstract)} \]
\[ i, x \in ide \]
\[ e, p \in exp \quad \text{(abstract)} \]
\[ \kappa \in store \rightarrow bool^T \]
\[ \gamma \in id \rightarrow store \rightarrow val^T \]
\[ \sigma \in store^T = ide \rightarrow val^T \]
\[ \delta \in update = store^T \rightarrow ide \rightarrow val^T \rightarrow store^T \]

Semantics:

\[ \mathcal{E} : \mathit{exp} \rightarrow store^T \rightarrow val^T \quad \text{(abstract)} \]
\[ \mathcal{C}' : \mathit{update} \rightarrow \mathit{stmt} \rightarrow ide \rightarrow (store^T \rightarrow bool^T) \rightarrow (ide \rightarrow store^T \rightarrow val^T) \]
\[ \rightarrow store^T \rightarrow val^T \]

\[ A_{\mathcal{C}'} : C'[x := e] = \lambda i.\kappa.\lambda.\sigma.\kappa.\sigma \rightarrow \delta(\sigma, x, \mathcal{E}[e](\lambda j.\gamma j \sigma)) i, \perp \]
\[ I_{\mathcal{C}'} : C'[\mathit{if} p s_1 s_f] = \lambda i.\kappa.\lambda.\gamma.\sigma \rightarrow \delta(\mathcal{A}(s_1, i) \vee \mathcal{A}(s_f, i)) \rightarrow \lambda.\sigma.\gamma.\sigma, \]
\[ \text{let } \kappa^+ = \lambda.\sigma.\kappa \wedge \mathcal{E}[p](\lambda j.\gamma j \sigma), \]
\[ \kappa^- = \lambda.\sigma.\kappa \wedge \kappa \wedge \mathcal{E}[p](\lambda j.\gamma j \sigma) \text{ in} \]
\[ (\mathcal{A}(s_1, i) \rightarrow (\lambda.\sigma.\kappa^+ \sigma \rightarrow \gamma i \sigma, \perp), C'[s_1][i \kappa^+] \perp) \cup \]
\[ (\mathcal{A}(s_f, i) \rightarrow (\lambda.\sigma.\kappa^- \sigma \rightarrow \gamma i \sigma, \perp), C'[s_f][i \kappa^-] \perp) \]

\[ C_{\mathcal{C}'} : C'[s_1; s_2] = \lambda i.\kappa.\lambda.\gamma.\sigma \rightarrow \delta(\lambda j.\sigma', \lambda j.\delta[s_1][j \kappa \gamma]) \]
\[ W_{\mathcal{C}'} : C'[\mathit{while} p s] = \text{fix}(\lambda w.\lambda i.\kappa.\gamma.\sigma \rightarrow \delta(\lambda j.\sigma', \perp), \lambda.\sigma.\gamma.\sigma, \lambda.\kappa.\gamma.\sigma) \]
\[ \text{let } \kappa^+ = \lambda.\sigma.\kappa \wedge \mathcal{E}[p](\lambda j.\gamma j \sigma), \]
\[ \kappa^- = \lambda.\sigma.\kappa \wedge \kappa \wedge \mathcal{E}[p](\lambda j.\gamma j \sigma) \text{ in} \]
\[ (\lambda.\kappa^- \sigma \rightarrow \gamma i \sigma, \perp) \cup (w i k^+(\lambda j.\sigma', \lambda.\kappa^+)) \]

\[ C_{\mathcal{C}'} : C'[s; \mathit{end}] = \lambda i.\kappa.\lambda.\kappa.\gamma.\sigma \rightarrow \delta(\lambda j.\sigma', \perp) \]

\[ \text{Figure 3.6 Abstraction Step} \]
Figure 3.7 Domains for Program Dependence Trees

must be separated into static component that does not require the initial store and
a dynamic component. The static identifier test is independent of the initial store.
In the case of a match on the identifier, the information required to evaluate the
assignment includes the expression e, a table of code containing the trees for each
identifier referenced in the expression e, and the code for \( \kappa \). The code table for an
expression, built by the function \( MS \), contains a finite number of entries since an
expression only references a finite number of identifiers. The function \( MS \) is defined
as follows:

\[ MS(e, \gamma) = \{(j, \gamma j) \mid j \in \mathcal{R}(e)\} \]

The function \( \mathcal{R} \) finds the set of identifiers referenced in an expression.

If the assigned identifier does not match the demanded identifier, the demand is
propagated through the \( \gamma \) argument. Thus, the \( G \) clause for lazy assignment,

\[ G_Z[x := e] = \lambda i \kappa \gamma.i \vdash x \rightarrow (\kappa, e, MS(e, \gamma)), \gamma i \]

builds a triple containing the required information if the identifiers match and prop-
agates the demand further otherwise.

The clause in \( P \) that interprets this triple,

\[ P[(\kappa, e, C_e)] = \lambda \sigma. P_e[\kappa] \sigma \rightarrow E[\varepsilon] \mathcal{RS}(C_e, \sigma), \bot \]
Semantics:

\( \mathcal{G} : \text{update} \to \text{stmt} \to \text{ide} \to p\text{-node} \to \text{code-tbl} \to \text{code} \)

\( \mathcal{E} : \text{exp} \to \text{store}^T \to \text{val}^T \quad \text{(abstract)} \)

\( \mathcal{P} : \text{code} \to \text{store}^T \to \text{val}^T \)

\( \mathcal{P}_k : p\text{-node} \to \text{store}^T \to \text{bool}^T \)

\[
\mathcal{G}_K[x := e] = \lambda \gamma. i \vdash z \to (\kappa, e, \mathcal{M}(e, \gamma), \mathcal{M}(i, \gamma)), \gamma_i
\]

\[
\mathcal{G}_Z[x := e] = \lambda \gamma. i \vdash z \to (\kappa, e, \mathcal{M}(e, \gamma)), \gamma_i
\]

\[
\mathcal{G}[\text{if } p \ s \ s_f] = \lambda \gamma. (\neg (A(s, i) \lor A(s_f, i))) \to \gamma_i,
\]

\[
\text{let } \kappa^+ = (T, \kappa, p, \mathcal{M}(p, \gamma)), \kappa^- = (F, \kappa, p, \mathcal{M}(p, \gamma)) \text{ in }
\]

\[
\neg A(s, i) \to \mathcal{M}(\kappa^+, i, \gamma), \mathcal{G}[s_i]i\kappa^+, \neg A(s_f, i) \to \mathcal{M}(\kappa^-, i, \gamma), \mathcal{G}[s_f]i\kappa^-,
\]

\[
\mathcal{G}[s_1; s_2] = \lambda \gamma. \mathcal{G}[s_2]i\kappa(\lambda j. \mathcal{G}[s_1]j\kappa)
\]

\[
\mathcal{G}[\text{while } p \ s ] = \text{fix}(\lambda w. \lambda \gamma. \neg A(s, i) \to \gamma_i,
\]

\[
\text{let } \kappa^+ = (T, \kappa, p, \mathcal{M}(p, \gamma)), \kappa^- = (F, \kappa, p, \mathcal{M}(p, \gamma)) \text{ in }
\]

\[
\mathcal{M}(\kappa^-, i, \gamma) \circ w \kappa^+ (\lambda j. \mathcal{G}[s]j\kappa^+)
\]

\[
\mathcal{G}[s; \text{end}] = \lambda \gamma. (i, \mathcal{G}[s]i\kappa)
\]

\[
\mathcal{P}[(\kappa, e, C_o)] = \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \to \mathcal{E}[e] \mathcal{R}(C_o, \sigma), \bot
\]

\[
\mathcal{P}[(\kappa, e, C_o, C_o)] = \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \to (\mathcal{P}[C_o] \sigma \downarrow \to \bot, \mathcal{E}[e] \mathcal{R}(C_o, \sigma)), \bot
\]

\[
\mathcal{P}[(C_o, C_f)] = \lambda \sigma. \mathcal{P}[C_f] \sigma \cup \mathcal{P}[C_f] \sigma
\]

\[
\mathcal{P}[(C, C_o, ...)] = \lambda \sigma. \mathcal{P}[C] \sigma \cup \mathcal{P}[(C_o, ...)] \sigma
\]

\[
\mathcal{P}[(\kappa, i)] = \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \to \sigma, \bot
\]

\[
\mathcal{P}[(i, c_j)] = \lambda \sigma. \mathcal{P}[c_j] \sigma
\]

\[
\mathcal{P}_k[\text{true}]= \lambda \sigma. T
\]

\[
\mathcal{P}_k[(T, \kappa, p, C_o)] = \lambda \sigma. \mathcal{E}[p] \mathcal{R}(C_o, \sigma) \land \mathcal{P}_k[\kappa] \sigma
\]

\[
\mathcal{P}_k[(F, \kappa, p, C_o)] = \lambda \sigma. \neg \mathcal{E}[p] \mathcal{R}(C_o, \sigma) \land \mathcal{P}_k[\kappa] \sigma
\]

Auxiliary Functions:

\[
\mathcal{M}(e, \gamma) = \{(j, \gamma j) \mid j \in \mathcal{R}(e)\}
\]

\[
\mathcal{M}(\kappa, i, \gamma) = (\kappa, i, \mathcal{M}(i, \gamma))
\]

\[
\mathcal{R}(C_o, \sigma) = \{(j, \mathcal{P}[C_f] \sigma) \mid (j, C_f) \in C_o\}
\]

Figure 3.8  Semantics for Program Dependence Trees
reconstructs the result from the information in the triple and the initial store. The function \( P_k \) determines the value of the control node \( \kappa \). If the value is true, the expression \( e \) is evaluated in a store built from the code table for expression \( e \). The function \( RS \) constructs a store for an expression from the code table for that expression and a store as follows:

\[
RS(C_e, \sigma) = \{(j, P[j][\sigma]) \mid (j, C_j) \in C_e\}
\]

Specifically, if constructs, for each identifier-code pair \((j, C_j)\) in the code table, a pair for the finite store required to evaluate the expression. The code table \( C_j \) is evaluated, using the initial store, to obtain the value for the identifier \( j \). The construction of the finite table that serves as the store is strict; this ensures that all dependences are resolved before the evaluation of the expression.

The clause in \( G \) for lackadaisical assignment,

\[
G_K[x := e] = \lambda i. \kappa. \gamma. \{ x \rightarrow (\kappa, e, MS(e, \gamma), MS(i, \gamma)), \gamma \}
\]

differs from the lazy assignment clause in the additional component for the output dependence. In addition to building trees for identifiers referenced in the expression, a tree to compute the previous value of the demanded identifier \( i \) is also built. The \( P \) clause for the output assignment tuple requires this value to be defined before expression evaluation begins.

The function \( G \) builds two different node types: data nodes and control nodes. A data node is a tuple as described above for the assignment statement. A control node encodes the control parameter information which is used to determine if a node is enabled. Control nodes are built by both the if and the while clauses. For example, the if clause,

\[
G[if p s_t s_f] = \lambda \kappa. \gamma. \neg(A(s_t, i) \vee A(s_f, i)) \rightarrow \gamma,
\]

let \( \kappa^+ = (T, \kappa, p, MS(p, \gamma)), \kappa^- = (F, \kappa, p, MS(p, \gamma)) \) in

\[
\neg A(s_t, i) \rightarrow MV(\kappa^+, i, \gamma), G[s_t][i, \kappa^+], \gamma
\]

\[
\neg A(s_f, i) \rightarrow MV(\kappa^-, i, \gamma), G[s_f][i, \kappa^-], \gamma
\]

builds control nodes \( \kappa^+ \) and \( \kappa^- \) for the true and false branches respectively. Each control node contains the predicate expression, a finite code table to evaluate the expression and the incoming control node. The code table is built in the same manner as for the assignment statement expression described above. The true control node is designated by the tag T; the false node by the tag F. The function \( P_k \) evaluates
these control nodes. The base is the starting value of $\text{true}_c$. The clause in $P_k$ for the $\text{true}$ node,

$$P_k[(T, \kappa, p, C_p)] = \lambda \sigma.\mathcal{E}[p] \mathcal{R}(C_p, \sigma) \land P_k[\kappa] \sigma$$

computes the logical and of the value of the prior control node $\kappa$ in the initial store and the value of the predicate expression $p$ computed in its finite store. The finite store for the predicate expression is constructed using the function $\mathcal{R}$. The if clause constructs a tree with the trees for the $\text{true}$ branch and the $\text{false}$ branch as the tree components. The $P$ function uses the least upper bound of the results of the two components as the result for the tree as a whole:

$$P[(C_t, C_f)] \sigma = P[C_t] \sigma \cup P[C_f] \sigma$$

The if clause also builds value nodes, using the function $\mathcal{M}$. The value nodes replace def-order dependence edges by ensuring that only one value reaches any use of an identifier. For this language, with only atomic data values, data nodes with identity assignments as the expression are sufficient to implement the value nodes. A value node is built in the if clause if the demanded identifier is defined on one branch of the if but not the other. The value node receives the control node built by the if clause, so the value node only returns a value if the definition on the other branch of the if statement is not enabled. The term value node is appropriate since the node serves to stop the data flow from before the if statement if that value is not needed.

The while clause builds a value node for each iteration; all values flowing out of a while loop go through value nodes. The valve node for the first iteration controls the flow of data from before the while loop, analogous to the valve node built for the if statement. The while clause in the $G$ function constructs a tree with an infinite number of branches. Each branch represents one possible execution of the loop, a branch for zero iterations, a branch for one iteration, etc. At most one of these branches applies, so the $P$ clause takes the least upper bound of the result of each of these branches.

Finally, the value for $\gamma_0$ in Equation 3.1 must be specified. The initial code tree must allow the $P$ function to access values from the initial store. Code from the initial code tree is only needed if there is a definition-free path for the identifier through the program to the use. A new node type, idef nodes [31, 48], designates this situation. The only information in the idef node is the identifier and the constant $\text{true}_c$ for a control node. The value for $\gamma_0$ then is

$$\gamma_0 = (\lambda i.(\text{true}_c, i))$$
The clause in $P$ interpreting this node is as follows:

$$P[(\kappa, i)] = \lambda \sigma. P_k[\kappa]\sigma \rightarrow \sigma i, \bot$$

If the control node is true, the value of the identifier is accessed in the initial store $\sigma$. For program dependence trees that are images of language $W$ programs, the control node in an `idef` node never evaluates to false.

3.2.5 Example Program Dependence Tree

The `pdt` for program $P$ with $z$ as the demanded identifier,

$$P \equiv x := 1; \text{if } x \neq y \text{ then } x := 3 \text{ else } y := 3; z := f(x); \text{end}$$

appears in Figure 3.9. Since there is no definition of $y$ before the use of $y$ in the

---

![Diagram](image)

**Figure 3.9** Program Dependence Tree for Program $P$
predicate expression, an **idf** node is required. The identifier \( x \) is defined only on the **true** branch of the **if** statement, so a valve node appears on the **false** branch. The definition for \( x \) outside the **if** sends its value to both the predicate expression and the valve node. The use of \( x \) after the **if** statement receives the value for \( x \) either from the valve node or from the assignment to \( x \) within the **if** statement. The **final** node for \( z \) serves as the output store for the **pdt**.

In Figure 3.9, there are multiple nodes for single statements in the program \( P \). For example, there are three copies of the assignment node for the statement \( x := 1 \). This duplication retains the tree property of the **pdt**. Each individual demand for an identifier generates a new **pdt** to compute that value, resulting in the duplication of program components in the **pdt**.

### 3.3 Operational Semantics for Program Dependence Trees

This section describes an operational semantics for the **pcts** described in the previous section. The operational semantics captures the programmer’s intuition about how evaluation proceeds in the context of the **pdt**. This semantics is a graph rewriting semantics [52], consisting of a set of rewriting rules for the nodes in the **pdt**.

The domain definitions in Figure 3.8 of the previous section specify the shape of the trees to which the operational semantics applies. While all the information is present in this representation, it is not a useful representation for the operational semantics. The **d-node** domain describes six different nodes: **set** nodes, **out** nodes, **idf** nodes, **final** nodes, **if** nodes and **while** nodes. The **if** and **while** nodes are composite nodes that only combine the effects of the component trees. The **p-node** domain defines three other node types: **start** nodes, **true** predicate nodes, and **false** predicate nodes. The function defined in Figure 3.10 specifies the mapping between the representation of dependence information as **pcts** and the infinite **pdt graphs** for which the operational semantics is defined.

Figure 3.11 specifies the information contained in each of the different node types. The domain definitions for the **pdt** itself are repeated in this figure, along with descriptive tags for each component.

Each **setz** tuple\(^9\) contains a predicate tuple, an expression component and a code table to evaluate the expression. The predicate tuple determines the incoming control

\(^9\)In the following discussion, **tuple** refers to a component of the input **pdt** and **node** refers to a component of the output **pdt graph**.
\( \mathcal{K}[\langle \kappa, j \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } \{ (i_{m'}, j, m) \} \cup \mathcal{K}[\kappa]_{\text{ism}'} \)

\( \mathcal{K}[\langle \kappa, e, c_d \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } \{ (s_{m'}, i := e, m) \} \cup \bigcup_{(i, c_j) \in e} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{ism}' \)

\( \mathcal{K}[\langle \kappa, e, c_e, c_x \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } s \triangleright T \rightarrow \{ (s_{m'}, i := e, m) \} \cup \bigcup_{(i, c_j) \in e} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{i Tm} \)

\( \bigcup_{(i, c_j) \in e} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{i Pm}' \),

\( \{ (s_{m'}, i := e, m) \} \cup \bigcup_{(i, c_j) \in e} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{i Tm} \)

\( \bigcup_{(i, c_j) \in e} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{i Pm}' \),

\( \mathcal{K}[\langle j, c_j \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } \{ (f_m', i) \} \cup \mathcal{K}[c_j]_{jsm'} \)

\( \mathcal{K}[\langle c_1, c_f \rangle] \) = \( \lambda \text{ism.} \mathcal{K}[c_j]_{jsm} \cup \mathcal{K}[\kappa]_{jsm} \)

\( \mathcal{K}[\langle c_1, c_2, \ldots \rangle] \) = \( \lambda \text{ism.} \mathcal{K}[c_1]_{jsm} \cup \mathcal{K}[\kappa]_{jsm} \)

\( \mathcal{K}[\langle \text{true}_e \rangle] \) = \( \lambda \text{ism.} \{ (p^e_{\text{gen}}, \text{true}_e, m) \} \)

\( \mathcal{K}[\langle T, \kappa, p, c_p \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } \{ (p^e_m', p, m) \} \cup \bigcup_{(i, c_j) \in p} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{ism}' \)

\( \mathcal{K}[\langle F, \kappa, p, c_p \rangle] \) = \( \lambda \text{ism.} \text{let } m' = \text{gen in } \{ (p^f_m', p, m) \} \cup \bigcup_{(i, c_j) \in p} \mathcal{K}[c_j]_{jsm'} \cup \mathcal{K}[\kappa]_{ism}' \)

**Figure 3.10** Mapping Function from Domain Element to Graph Node
Semantic Domains:

c_σ ∈ code-tbl = ide → code

\( c ∈ \text{code} = d\text{-node} \oplus (d\text{-node} \otimes d\text{-node}) \oplus = d\text{-node}^+ \)

\( c_d ∈ d\text{-node} = (p\text{-node} \otimes \exp \otimes \text{code-tbl}) \oplus (p\text{-node} \otimes \text{idc}) \oplus (p\text{-node} \otimes \exp \otimes \text{code-tbl} \otimes \text{code-tbl}) \oplus = (\text{ide} \otimes \text{code}) \)

\( c_κ ∈ p\text{-node} = \text{true}_c \oplus (p\text{-node} \otimes \exp \otimes \text{code-tbl}) \oplus (p\text{-node} \otimes \exp \otimes \text{code-tbl}) \)

\[
\begin{align*}
\text{idef} & = (i_m, x) \\
\text{set} & = (s_m, x := e) \\
\text{out} & = (s_m, x := e) \\
\text{start} & = (p_m^p, \text{true}_c) \\
\text{true} & = (p_m^l, e) \\
\text{false} & = (p_m^f, e) \\
\text{final} & = (f_m, x)
\end{align*}
\]

Figure 3.11 Representations for PDT Nodes

dependence, while the code table determines the incoming data dependences. The identifiers in the code table become labels on the incoming data dependences. In the case of the lackadaisical update function and set_κ, the data dependences are divided between output and flow dependences. Therefore, there are both set nodes and out nodes in the tree; a node is a set node if it appears in the code-table component for an expression. If the node appears in the output component of a data-node tuple, the node is an out node. For predicate tuples, there are two different cases to consider. Predicate tuples for true_c are equivalent to start nodes in the pdt graph. The true and false predicate tuples each contain a predicate tuple, a predicate expression and a code table. As with the set tuple, the code table specifies the incoming data dependences and the predicate tuple specifies the incoming control dependence for the predicate node. The final tuples contain an identifier and the code specifying the value of that identifier. The code defines the incoming data dependences for the final node. Finally, idef tuples contain a predicate tuple and an identifier. For pdts, the start tuple is always the predicate tuple for an idef tuple. There is no expression in the idef tuple since this tuple extracts the value for the identifier from the initial store.
While the tree formulation naturally specifies a predecessor relation, the rewriting semantics uses the corresponding successor relation. Indeed, since the structure is a tree, the successor relation is a function. The successor function defines the edge sets typically associated with pdgs. Successor edges of predicate nodes are control dependence edges; successor edges of set nodes are flow dependences; successor edges of out nodes are output dependences. The tag on the predicate node is the label for the control dependence edge; edges from start nodes are assumed to have tag $T$.

To accommodate the operational semantics, unique labels are added to each of the nodes; the label of a node contains its node type. A pdt graph is a tuple $G = (N, S, \alpha)$. The function $S$ is the successor function for pdt graph $G$. The function $\alpha$ maps node labels to the expression component of the corresponding tuple. In the case of set nodes, the $\alpha$ function includes the identifier label from the outgoing data dependence as well. Thus, a set node $m$ is represented as a tuple $(s_m, x := e)$ where the label $s_m$ designates the node $m$ is a set node and $\alpha(m) = x := e$. For final and idf nodes, the $\alpha$ function specifies the identifier for the corresponding tuple. The node set $N$ is the disjoint union of node sets $N_I, N_S, N_P, N_F$ for the set of idf nodes, set nodes, predicate nodes and final nodes respectively. For pdt graph $G = (N_I, N_S, N_P, N_F, S, \alpha)$, the name of the pdt graph is also used to represent the union of the individual node sets. The set of predicate nodes is the disjoint union of the sets of start nodes, true nodes, and false nodes, and the set of set nodes is the disjoint union of the set and out nodes respectively.

The rewriting semantics for pdt graphs consists of a set of reduction rules for transforming pdt graphs relative to an initial store. The rules operate only on redexes in the tree. A redex in a pdt is a node $n$ such that no other node $m$ in the pdt graph depends on it.

**Definition 3.1 (Redex and Redex Set)** In pdt $G = (N_I, N_S, N_P, N_F, S, \alpha)$ node $n$ is a redex if and only if

$$\forall m \in G. S(m) \neq n$$

The notation $R(G)$ denotes the set of all redexes in pdt graph $G$.

Since a redex has no dependence predecessors, all dependence requirements for this node have been met. Thus, the node is enabled for evaluation and all the values required for expression evaluation are available.
There are different rewriting rules for each node set. In all cases, the rewriting rules remove the redex. Additional nodes may also be removed in the rewriting, and some node may receive a value for an identifier. The removal of nodes and edges corresponds to the resolving of the associated dependences. The substitution operation corresponds to the flow of data over the flow dependence edges. The rules for rewriting redex \( r \) in \( pdt \) graph \( G = \langle N_I, N_S, N_P, N_F, S, \alpha \rangle \) to \( pdt \) graph \( G' = \langle N'_I, N'_S, N'_P, N'_F, S', \alpha' \rangle \) relative to store \( \sigma \) are presented in Figure 3.12 with the definition of the substitution operation appearing in Figure 3.13. The new successor function is the restriction of the old function to the new node sets. The substitution operation, as shown in the figure, updates the labeling function for the affected nodes. Because the successor relation is actually a function, the result of a rewriting step maps back to a domain element which corresponds to the \( pdt \) with the particular dependence resolved.

The rewriting of a set node,

\[
\langle s_r, x := e \rangle, n = S(r) \left\{ \begin{array}{l}
N'_I = N_I \\
N'_S = (N_S - \{ r \})[n \leftarrow x/\mathcal{E}[e]] \\
N'_P = N_P[n \leftarrow x/\mathcal{E}[e]] \\
N'_F = N_F[n \leftarrow x/\mathcal{E}[e]]
\end{array} \right.
\]

propagates the value computed for the expression \( e \) to node \( n \), if the value of the expression is defined. The rewriting diverges if the expression denotes \( \perp \). The substitution operation \( j[x/v] \) specifies that the value \( v \) is substituted into the expression of \( j \) wherever the identifier \( x \) appears. For a set of nodes, the substitution applies only to the specified node. The expression substitution operation is unspecified since the expression language is unspecified. For example, the rewriting of node \( s_P \),

\[
\langle s_P, x := e \rangle
\]

\[ \xrightarrow{f} \]

\[ Q \xrightarrow{Q[x/v]} \]

if \( \mathcal{E}[e] = v \)

replaces all occurrences of \( x \) in the expression portion of the node \( Q \) with the value \( v \). The redex \( P \) is removed from the \( pdt \) graph in the rewriting, which removes the flow dependence for \( Q \).
\[
\begin{align*}
(i_r, x), S(r) & \quad \left\{ 
\begin{array}{l}
N'_I = N_I - \{r\} \\
N'_S = N_S[n \leftarrow x/\sigma x]
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_P = N_P[n \leftarrow x/\sigma x]
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_F = N_F[n \leftarrow x/\sigma x]
\end{array}
\right. \\
(s_r, x := n), S(r) & \quad \left\{ 
\begin{array}{l}
N'_I = N_I \\
N'_S = (N_S - \{r\})[n \leftarrow x/\epsilon[e]]
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_P = N_P[n \leftarrow x/\epsilon[e]]
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_F = N_F[n \leftarrow x/\epsilon[e]]
\end{array}
\right. \\
(s^g, x := e), S(r) & \quad \left\{ 
\begin{array}{l}
N'_I = N_I \\
N'_S = N_S - \{r\} \text{ where } \epsilon[e] \neq \bot
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_P = N_P
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_F = N_F
\end{array}
\right. \\
(p^b_r, e), S(r) & \quad \left\{ 
\begin{array}{l}
N'_I = N_I - Q_I
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_S = N_S - Q_S
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_P = N_P - Q_P - \{r\}
\end{array}
\right. \\
& \quad \left\{ 
\begin{array}{l}
N'_F = N_F - Q_F
\end{array}
\right. \\
\end{align*}
\]

where
\[
Q = \begin{cases}
\emptyset & \text{if } \epsilon[e] = b \lor e \equiv \text{true}_c \\
\{ m \mid (p^b_r, e) \in Q, m = S(q) \} \cup \{ n \} & \text{if } \epsilon[e] \neq b
\end{cases}
\]

\[
Q_i = \{ m \mid m \in Q \land m \in N_i \} \text{ for } i \in \{ I, S, P, F \}
\]

**Figure 3.12** Rewriting Rules for the Core Language

\[
N[m - x/v] = \begin{cases}
N & \text{if } m \notin N \\
N - \{m\} \cup \{m[x/v]\} & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
(i_m, x)[y/v] &= (i_m, x) \\
(s_m, x := e)[y/v] &= (s_m, x := e[y \leftarrow v]) \\
(p^b_m, e)[y/v] &= (p^b_m, e[y \leftarrow v]) \\
(f_m, x)[y/v] &= (s_m, x) \quad \text{if } x \neq y \\
(f_m, x)[x/v] &= (s_m, x[x \leftarrow v])
\end{align*}
\]

where \( e[x \leftarrow v] \) is the expression substitution operation.

**Figure 3.13** Substitution Operation for Nodes
The rewriting of an `ifdef` node proceeds as for an assignment node except that the value is extracted from the initial store instead of being computed by the expression evaluator. The rewriting of an `out` node removes only the redex and the output dependence; no value propagation is performed.\(^\text{10}\)

The rewriting of a predicate node,

\[
\langle p^b, e, \rangle, n = S(r) = \begin{cases} 
N^I_i &= N_i - Q_i \\
N^S_i &= N_S - Q_S \\
N^P_i &= N_P - Q_P - \{r\} \\
N^F_i &= N_F - Q_F 
\end{cases}
\]

where

\[
Q = \begin{cases} 
\emptyset & \text{if } E[e] = b \lor e \equiv \text{true}_c \\
\{m \mid \langle p^y_i, e' \rangle \in Q, m = S(q) \cup \{n\}\} & \text{if } E[e] \neq b 
\end{cases}
\]

\[
Q_i = \{m \mid m \in Q \land m \in N_i\} \text{ for } i \in \{I, S, P, F\}
\]

depends on whether the value of the predicate expression matches the tag in the node. If the value of the predicate matches the tag or if the expression is the constant `true_c`, representing the start node control dependences, only the redex is removed from the `pdt graph`. Otherwise, the node \(n\) is removed as are any children of predicate nodes in the sub-tree rooted at \(n\). This step removes any nodes that are transitively control dependent on node \(r\); the set \(Q\) contains all of these nodes. The sets \(Q_i\) partition \(Q\) based on node type. For example, the following picture demonstrates the rewriting of the redex \(p_N\) which is a `false` control dependence predecessor for node \(p_R\), assuming the predicate expression \(e\) evaluates to `true`:

\[
\langle p^I_N, e \rangle \rightarrow p_R \rightarrow d_S \rightarrow T \rightarrow T
\]

\(^{10}\)This rewriting rule demonstrates the control character of the output dependence, in that the only effect of this dependence is to sequence evaluations that are functionally unrelated.
The node $R$ is removed along with its control dependence successor, $S$. Node $T$ is not removed because it is a data dependence successor of $S$. The removing of the transitive control dependence successors is needed in this step, since the true control dependence relation is the transitive closure of the edges in the pdt graph.

These single step rules form a relation, the sequential rewriting relation, over pdt graphs.

**Definition 3.2 (Sequential Rewriting Relation)** Given a pdt graph, $G$, an initial store $\sigma$, and a redex $r$, the new pdt graph, $G'$ is found by applying the one-step sequential rewriting rules found in Figure 3.12. The sequential rewriting relation for redex $r$ is denoted $\xrightarrow{r,\sigma}$. The reflexive, transitive closure of the sequential rewriting relation is denoted $\xrightarrow{\ast}$. Redex selection for rewriting is arbitrary.

Applying the sequential rewriting relation to a set of redexes simultaneously defines the parallel rewriting relation:

**Definition 3.3 (Parallel Rewriting)** Let $G$ be a pdt graph, let $s = \{r_1, \ldots, r_n\} \subseteq R(G)$, be an arbitrary subset of $R(G)$, and let the graphs $G_i$ for $i = 1, \ldots, n$ be the result of sequentially rewriting redex $r_i$ in $G$ relative to the store $\sigma$,

$$G \xrightarrow{r_i} G_i.$$  

Then, pdt graph $G = (N_I, N_S, N_P, N_F, \alpha)$ parallel rewrites to pdt graph $G' = (N'_I, N'_S, N'_P, N'_F, \alpha')$ with respect to $s$ and $\sigma$, notation $G \xrightarrow{s,\sigma}_{P} G'$ if for $k \in \{I, S, P, F\}$,

$$N'_k = N_k - \bigcup(N_k - N_{k_i}) - Q_k \cup Q'_k$$

where

$$Q_k = \{q \mid q \in \bigcap N_{k_i} \land \exists x. C(q, x, s)\}$$

$$Q'_k = \{q[x_1/\top, \ldots, x_n/\top] \mid q \in Q_k \land C(q, x_i, s)\}$$

$$C(q, x, s) = \exists j, k \in s.(id(r_j) = id(r_k) = x) \land (S(j) = S(k) = q \land \text{val}(r_j) \neq \text{val}(r_k))$$

The reflexive, transitive closure of the parallel rewriting relation with respect to the store $\sigma$ is denoted $\xrightarrow{\ast}_{P}$.\[11\]

---

\[11\] We frequently drop the $\sigma$ from the notation since the store is never modified by the rewriting.
The conflict predicate, \( C(p, x, s) \), identifies nodes that received unequal values for identifier \( x \) from different redexes in the redex set \( s \). The selector \( id(r) \) denotes the identifier portion of either an \( \text{idef} \) or an assignment node while \( \text{val}(r) \) denotes the value of the expression component of node \( r \). The value for the identifier \( x \) in the substitution for the result of the parallel rewriting is the value \( T \). The sets \( Q_k \) contain all nodes that were not removed in the rewriting of some redex that satisfy the conflict predicate for some identifier. These nodes, with the \( T \) values substituted for any identifier receiving conflicting values, appear in the sets \( Q'_k \).

**Evaluation Relations**

The rewriting relations from the previous section induce evaluation relations as follows:

**Definition 3.4 (Evaluation Relations)** For \( \text{pdt graph} \ G \) and store \( \sigma \),

\[
\text{Eval}_s(G, \sigma) = \{ v \mid \exists m \quad G \xrightarrow{\sigma}^* (f_m, v) \}
\]

and

\[
\text{Eval}(G, \sigma) = \{ v \mid \exists m \quad G \xrightarrow{\sigma}^p (f_m, v) \}
\]

\( \text{Eval}_s \) and \( \text{Eval} \) correspond to the sequential and parallel rewriting relations respectively. Neither of these relations are functions in general. However, both are functions over \( \text{pdt graphs} \) that are the images of programs in \( \mathcal{W} \):

**Theorem 3.2** \( \text{Eval}_s \) and \( \text{Eval} \) are functions for \( \text{pdt graphs} \) that are images of programs in \( \mathcal{W} \) under \( G \).

**Proof** Since programs are deterministic, the conflict set will always be empty and control edges are as needed so there is only one result.

### 3.3.1 Rewriting Example

Figure 3.14 through Figure 3.19 show the rewriting of the example program \( Q \) from the previous section. In this example, all redexes are rewritten as soon as possible, and all available redexes are rewritten at each parallel step. The \( \text{pdt graph} \) of Figure 3.14 is the starting point. In these figures, we show the tree as a graph to simplify the presentation. In the first rewriting step, the \text{start} node is the only redex, so this node
Figure 3.14  Pdt Graph for Program $Q$
Figure 3.15  \textit{Pdt} Graph for Program \textit{Q} After 1 Rewriting Step
is rewritten. Figure 3.15 shows the graph after this step. The redexes at this point are the `ifdef` node for $y$ and the assignment node for $x$. Assuming $y$ has the value 1 in the initial store, the result of rewriting these two nodes is the `pdg` in Figure 3.16. The value 1 is propagated to the `if` node for both $x$ and $y$. The value 1 is also sent to the valve node for $x$ from the assignment node. The redexes and all their outgoing flow edges are removed from the graph. The only redex now is the `if` node. Since the predicate expression is `true`, the valve node on the `false` branch is removed with all its outgoing edges. The `if` node with its outgoing edges is also removed. The result of this step is in Figure 3.17. The assignment to $x$ is now the redex. The value 4 is propagated to the assignment node for $z$. The next step is handled the same. Figures 3.18 and 3.19 show these steps. The final result is a graph with no edges and a `final` node for the identifier $z$. The final value for $z$ is the value of the expression $v$ where $v = f(3)$.

---

**Figure 3.16**  *Pdt* Graph for Program $Q$ After 2 Rewriting Steps
Figure 3.17  *Pdt* Graph for Program *Q* After 3 Rewriting Steps

Figure 3.18  *Pdt* Graph for Program *Q* After 4 Rewriting Steps

Figure 3.19  Final *Pdt* for Program *Q*
3.4 Soundness of Evaluation

In this section, we show soundness of evaluation for the operational semantics for program dependence trees with respect to the denotational semantics. Then, we discuss the adequacy of the operational semantics.

**Theorem 3.3 (Soundness of Evaluation)** For program $M$ in language $\mathcal{W}$, identifier $i$ and store $\sigma$, if

$$
\text{Eval}(G[M][i(\text{true}_e)\gamma_0,\sigma]) = v
$$

then

$$
\mathcal{P}[G[M][i(\text{true}_e)\gamma_0][\sigma]) = v
$$

where $v$ is in the domain $\text{val}$.

**Proof** The theorem is obvious using Lemma 3.2, shown below. □

**Lemma 3.2** For pdt $g$ and its corresponding pdt graph $\bar{g}$ and store $\sigma$,

1) $\bar{g} \xrightarrow{\text{r}\sigma} (i,v) \supset \mathcal{P}[g][\sigma] = v$

2) $\bar{g} \xrightarrow{\text{r}\sigma} \emptyset \supset \mathcal{P}[g][\sigma] = \bot$

3) $\bar{g} \xrightarrow{\mu\sigma} \bar{g}' \supset \mathcal{P}[g][\sigma] = \mathcal{P}[g'][\sigma]$

**Proof** The proof proceeds by induction on the length of the longest path in $g$ and on the structure of $g$. It requires that a Substitution Lemma be provable for the expression language, which we assumed to hold in Section 3.1. □

While the operational semantics is sound with respect to the denotational semantics, the semantics is not adequate. The problem in the adequacy arises from the data-driven nature of the operational semantics and the demand-driven nature of the denotational semantics. The operational semantics corresponds to the computational model of most computer systems in that it evaluates all components that are encountered and ready to evaluate. The denotational semantics only evaluates nodes that are demanded. The following program illustrates the difference in the two semantics:

$$
z := 1/0; \text{ if } p \ x := 1 \ x := f(z)
$$
In this program, the value of the identifier $z$ is only required if the predicate $p$ is false. However, a data-driven semantics begins the evaluation of the assignment to $z$ because all its inputs are available. Even though the value is ultimately not required, the operational semantics does not detect that; thus, the rewriting of the program is undefined. However, the denotational semantics does assign a meaning to this program. This characteristic of the semantics identifies the essential difference between data driven and demand driven evaluation; Section 8.1.6 of the final chapter explores this difference further.
Chapter 4

Semantics of Dependence for Arrays

Language \( \mathcal{W} \) treats all data values as if they were atomic data values. In this chapter, the language is extended to allow specifically for structured data items, arrays in particular. Arrays are used extensively in scientific computation, and the focus of much of the work in parallelization is computing with arrays. While the semantics of the previous chapter accommodates arrays with no changes, the dependences that arise under this interpretation force a serialization of all accesses to an array. This dependence relation is too restrictive to be useful. In this chapter, we introduce the notion of a partial array which retains the data flow character of the \( \text{pdt} \) but restricts the dependence relation based on the accesses to particular indices within the array.

The operations required to support arrays are array accesses and array updates. Both accesses and updates are specified through the name of the array and an expression whose value designates the desired index in the array. Array accesses appear in expressions; array updates are statements included in the syntax of the language itself. The first section describes the complications in data dependence analysis that arise in the presence of arrays. The next section presents a denotational semantics for the language \( \mathcal{W} \) augmented with explicit array values. The denotational and operational semantics for \( \text{pdt} \)s supporting arrays are described in the final sections.

4.1 Program Dependences and Arrays

The addition of arrays to the language affects the data dependence relation but not the control dependence relation. Data dependence analysis, however, is complicated by arrays. Several approaches to dealing with arrays have been proposed; the approaches fall into two general categories.

The first approach, referred to as the functional approach, treats the entire array as a single data value. The semantics of the previous chapter can accommodate array values using this approach. The advantage of this approach is the simplicity of the dependence analysis; no change is required to the analysis techniques used for simple
identifiers. There are two primary disadvantages to the functional approach. First, treating the array as a single data value requires that all updates to the array be serialized, even if they refer to distinct indices. In addition, any access to the array must wait for all prior updates to complete, even if the updates do not affect the accessed index. This approach severely restricts the amount of parallelism available in array accesses. The second disadvantage of this approach is the amount of copying required to transmit the array value to the different uses. In a data flow setting, the entire array structure flows to each use of the array. Optimizations that create temporary copies of data values also suffer under this approach since the amount of memory used as well as the time to copy the structure is excessive for most arrays.

The second approach, referred to as the shared memory approach, treats an array as a block of memory. The primary advantage of this approach is the efficiency of the access and update operations. Only one copy of the array exists; updates are made in place and all accesses refer to the same copy of the array. This approach closely models the implementation of arrays on conventional machines and exposes the memory accesses to allow for some access-related optimizations. The disadvantage of this approach is the complexity of the data dependence analysis [7]. Analysis of subscript expressions to determine overlapping accesses is in general undecidable. In practice, conservative, heuristic estimates are used; computing these estimates is a complicated process. In addition to the dependences required for atomic data, additional storage related dependences are also introduced. Even with a data flow approach, if the array is modeled as a block of memory, output dependences and anti-dependences are required to maintain the correctness of the program. However, dependence tests have been developed to detect these dependences and remove much of the serialization required by the functional approach.

One compromise approach, i-structures [7], are write-once structures that can be used to implement arrays. While these structures are closer in spirit to the data flow model, many of the disadvantages of the functional approach also apply to with i-structures. In addition, i-structures are not yet supported in traditional scientific computing languages.

Our approach, detailed in Section 4.3, is a compromise between the previous approaches. We retain the idea of an array as an aggregate object, preserving the data flow perspective, but we use the available dependence analysis to build partial arrays containing only those indices that may be required to satisfy a given access. These partial arrays are the aggregate objects that flow between statements, as opposed
to the full array. This approach provides a simple characterization of dependence in the presence of arrays, and isolates the additional complexities of the subscript dependence analysis. In addition, the pdt specified here does not require any anti-dependences to represent the program. Building on this approach, we can derive a model that does support the shared memory interpretation of arrays. The additional storage dependences are derivable from the data dependences present in the pdt.

4.2 Language Semantics for Arrays

The syntax for the array language, includes an update statement for arrays:

\[
\begin{align*}
  x & \in \text{id}_s \\
  a & \in \text{id}_a \\
  e & \in \text{exp} \\
  s & ::= \text{if } e \text{ s s while } e \text{ s x } := e | s; s | a[e] := e \\
  P & ::= s; \text{end}
\end{align*}
\]

In the array update statement,

\[a[e] := e'\]

\(a\) ranges over array identifiers, and \(e\) and \(e'\) are expressions. For simplicity, we assume that the set of array identifiers, \(\text{id}_a\), and the set of simple identifiers, \(\text{id}_s\), are disjoint. The rest of the language remains the same, except that expressions may include array accesses in addition to references to simple identifiers. The expression

\[a[e]\]

represents the value stored in index \(v\) of array \(a\), if the value of the expression \(e\) is the value \(v\). Conceptually, arrays are finite functions that map index values to values; an array is really a store within a store. Array updates alter the array function for the particular index value of the update. Array accesses are applications of the array function to the value of the index expression.

4.2.1 Denotational Semantics of the Array Language

The denotational definition for the array language, shown in Figure 4.1, is quite similar to that for language \(\mathcal{W}\). Arrays denote functions from values to values. The first value parameter is the index within the array that is desired. The clause for the
array update statement, U, updates the store with the new value for array a. The value of the index expression e' and the value expression e are both required for this update. The type of the store update function becomes

\[
\text{store} \rightarrow \text{ide} \rightarrow \text{val} \rightarrow \text{val} \rightarrow \text{store}
\]

to support arrays since the index value must be supplied in addition to the identifier. The clause for simple assignment, A, changes to accommodate the new update function. Any value would suffice in the index position since array identifiers and simple identifiers are disjoint. We use 0 for this dummy index value. The other clauses in the semantics are unchanged from the previous section.

### 4.3 Program Dependence Trees with Arrays

As with the earlier derivation, we begin with a standard denotational definition for our language with arrays; Figure 4.1 from the previous section contains this semantics. Figure 4.2 presents the demand stage of the semantics, with the identifier and index parameters moved forward from the output store in the function definition. The assignment and array update clauses in the figure have been expanded using the lazy store update function.
Semantic Domains:

\[
\begin{align*}
    b, v & \in \text{val} \quad \text{(abstract)} \\
    x & \in \text{ides} \\
    a & \in \text{ide}_a \\
    i & \in \text{ide} = \text{ide}_a \oplus \text{ide}_s \\
    e, e', p & \in \text{exp} \\
    \sigma & \in \text{store} = \text{ide} \rightarrow \text{val} \rightarrow \text{val}
\end{align*}
\]

Semantics:

\[
\begin{align*}
    \mathcal{E} : \text{exp} & \rightarrow \text{store} \rightarrow \text{val} \quad \text{(abstract)} \\
    \mathcal{V} : \text{stmt} & \rightarrow \text{ide} \rightarrow \text{val} \rightarrow \text{store} \rightarrow \text{val}
\end{align*}
\]

\[
\begin{align*}
    \text{Av} & : \forall[x := e] = \lambda \text{ib}.i \in x \rightarrow \mathcal{E}[e] \sigma, \sigma i b \\
    \text{Uv} & : \forall[a[e'] := e] = \lambda \text{ib}.\mathcal{O}(a,e',i,b) \rightarrow ((a,\mathcal{E}[e] \sigma) \in (i,b) \rightarrow \mathcal{E}[e] \sigma, \sigma i b, \sigma i b)
\end{align*}
\]

**Figure 4.2** Data Stage for Array Semantics

The array update clause, \( \text{Uv} \), includes a reference to the subscript analysis oracle \( \mathcal{O}_p \), that is specific to the program being analyzed. The oracle test is analogous to the equality test in the atomic assignment clause and represents the array data dependence patterns in the program. The predicate \( \mathcal{O}_p(a,e',i,b) \) is true if and only if \( a \) and \( i \) are the same array and subscript-analysis determines that subscript expression \( e' \) may represent the same array index as the demanded index value \( b \). An implicit decoration of the array update information is included to specify for the oracle for which points in the program the oracle is being queried. The program point of both the demand and the update is relevant to the oracle. To make this decoration explicit, the program point (statement number along with an iteration vector specifying which iteration relevant loops apply) of both the demand and the array update statement become parameters to the oracle function. In addition, the program point of the demand is an additional parameter to the semantic function as a whole. This information is not included in the semantic function to simply the description of the clauses. At the extreme, a function that always returns \text{true} suffices as an oracle. Using this oracle is equivalent to the functional approach to arrays described earlier.

The denotation for array updates thus contains a static and dynamic component. The oracle test, the static portion, determines if this update may represent the demanded value. If this test fails, the demand is immediately passed to the store, \( \sigma \).
Otherwise, the actual update index is computed from the expression \( e' \) and compared to the value \( s \). The index propagated in the \( V \) function will always be a value, not an expression. If this update operation does satisfy the demand, the value of the expression \( e \) is returned; otherwise the request is passed to the store, \( \sigma \). The oracle removes from consideration those array updates that can not affect the demanded index, based on the static analysis. The other clauses remain the same, with the addition of the index parameter \( b \) in the store accesses and in the definition of \( V \) itself. For example, the assignment clause, \( A_V \), includes the index parameter \( b \). This parameter is ignored if the assignment is for the demanded simple identifier \( i \). Otherwise, \( b \) is passed to the store along with \( i \) to further propagate the demand.

The relationship between the \( W \) and the \( V \) semantic functions relies on the notion of a conservative dependence test.

**Definition 4.1 (Conservative Dependence Oracle)** An oracle \( O_P \) for array dependence testing in program \( P \) is conservative if

\[
((a, E[e']\langle \sigma \rangle) = (i, E[b]\langle \sigma \rangle) \land E[b]\langle \sigma \rangle \neq \perp) \supset O_P(a, e', i, b) = T
\]

for all stores \( \sigma \), identifiers \( a \) and \( i \), expression \( e' \) and value \( b \).

With this definition of a conservative oracle, the following theorem establishes the correspondence between the semantic functions \( W \) and \( V \).

**Theorem 4.1** The semantic functions \( W \) and \( V \) satisfy the invariant

\[
(W[P] delta \sigma)_i s = V[P] i s \sigma
\]

for program \( P \), store \( \sigma \), identifier \( i \), and any index \( s \), if the array dependence oracle \( O_P \) is conservative.

**Proof** The proof proceeds by induction on the structure of the program \( P \). All cases other than the array update clause proceed as for language \( W \). The applicable clauses for an array update are

\[
W[a[e'] := e] \sigma = \delta(\sigma, a, E[e']\langle \sigma \rangle, E[e] \langle \sigma \rangle)
\]

\[
= (a, E[e'] \langle \sigma \rangle) \xrightarrow{2} (i, b) \rightarrow E[e] \langle \sigma \rangle, \sigma ib
\]

and

\[
V[a[e'] := e] ib \sigma = O_P(a, e', i, b) \rightarrow ((a, E[e'] \langle \sigma \rangle) \xrightarrow{2} (i, b) \rightarrow E[e] \langle \sigma \rangle, \sigma ib), \sigma ib
\]
If the oracle query returns \texttt{false}, the result is immediate since the oracle is conservative by assumption. Otherwise, the clause in the function \( V \) is simply the unrolling of the lazy update function for stores with arrays, and the result is again obvious.

\[ \square \]

### 4.3.1 Control Stage for Arrays

The addition of the control parameter \( \kappa \) is shown in Figure 4.3. As in the semantics for language \( \mathcal{W} \), the \texttt{if} and \texttt{while} clauses determine if the demanded identifier is assigned

---

**Semantic Domains:**

\[
\begin{align*}
b, v & \in \text{val}^T = \text{val} \cup \{ \top \} \\
x & \in \text{ide}_a \\
a & \in \text{ide}_a \\
i & \in \text{ide} = \text{ide}_a \oplus \text{ide}_a \\
e, e', p & \in \text{exp} \quad \text{(abstract)} \\
\kappa & \in \text{bool}^T \\
\sigma & \in \text{store}^T = \text{ide} \rightarrow \text{val}^T \rightarrow \text{val}^T \rightarrow \text{val}^T
\end{align*}
\]

**Semantics:**

\[
\begin{align*}
\mathcal{E} : \text{exp} & \rightarrow \text{store}^T \rightarrow \text{val}^T \quad \text{(abstract)} \\
\mathcal{C} : \text{stmt} & \rightarrow \text{ide} \rightarrow \text{val}^T \rightarrow \text{bool}^T \rightarrow \text{store}^T \rightarrow \text{val}^T
\end{align*}
\]

\[
\begin{align*}
\mathcal{A}_C : \mathcal{C}[x := e] & = \lambda \mathcal{ib} \mathcal{k} \mathcal{a}. \mathcal{i} \vdash x \rightarrow (\kappa \rightarrow \mathcal{E}[e] \sigma, \bot), \sigma \mathcal{i} \mathcal{b} \\
\mathcal{I}_C : \mathcal{C}[\text{if } p \; s_1 \; s_f] & = \lambda \mathcal{ib} \mathcal{k} \mathcal{a} . \neg (\mathcal{OC}(s_1, i, b) \lor \mathcal{OC}(s_f, i, b)) \rightarrow \sigma \mathcal{i} \mathcal{b}, \\
& \quad \text{let } \kappa^T = \kappa \land \mathcal{E}[p] \sigma, \kappa^- = \kappa \land \neg \mathcal{E}[p] \sigma \text{ in} \\
& \quad (\neg \mathcal{OC}(s_1, i, b) \rightarrow (\kappa^T \rightarrow \sigma \mathcal{i} \mathcal{b}, \bot), \mathcal{C}[s_1] \mathcal{ib} \kappa^T \sigma) \cup \\
& \quad (\neg \mathcal{OC}(s_f, i, b) \rightarrow (\kappa^- \rightarrow \sigma \mathcal{i} \mathcal{b}, \bot), \mathcal{C}[s_f] \mathcal{ib} \kappa^- \sigma) \\
\mathcal{U}_C : \mathcal{C}[a[e'] := e] & = \lambda \mathcal{ibk} \mathcal{a} . \mathcal{O}(a, e', i, b) \rightarrow (\kappa \rightarrow ((a, \mathcal{E}[e'] \sigma) \vdash (i, b) \rightarrow \mathcal{E}[e] \sigma, \sigma \mathcal{i} \mathcal{b}), \bot), \sigma \mathcal{i} \mathcal{b} \\
\mathcal{W}_C : \mathcal{C}[\text{while } p \; s] & = \text{fix}(\lambda w. \lambda \mathcal{ibk} \mathcal{a} . \neg \mathcal{OC}(s, i, b) \rightarrow \sigma \mathcal{i} \mathcal{b}, \\
& \quad \text{let } \kappa^T = \kappa \land \mathcal{E}[p] \sigma, \kappa^- = \kappa \land \neg \mathcal{E}[p] \sigma \text{ in} \\
& \quad \mathcal{wibk}^T (\lambda \mathcal{j} \mathcal{c} \mathcal{C}[s] \mathcal{jck}^T \sigma) \cup (\kappa^- \rightarrow \sigma \mathcal{i} \mathcal{b}, \bot)) \\
\mathcal{C}_C : \mathcal{C}[s; \text{end}] & = \lambda \mathcal{ibk} \mathcal{a} . \mathcal{C}[s] \mathcal{ibk} \mathcal{a}
\end{align*}
\]

**Figure 4.3** Control Stage of Array Derivation
within the body of the if statement before interpreting the predicate expression. However, to support both simple and array identifiers, a new predicate, \( OA \), is used. The definition for \( OA \) appears in Figure 4.4. The predicate \( OA(s, i, b, \mathcal{O}_P) \) is true if and only if \( i \) is a simple identifier and \( i \) is assigned in \( s \), or if \( i \) is an array identifier and there is some array update statement \( a[e'] := e \) in \( s \) such that \( \mathcal{O}_P(a, e', i, b) \) is true. This predicate is the logical extension of the \( A \) predicate, used in the previous chapter, incorporating the subscript analysis oracle.

The array update clause includes a reference to the control parameter, \( \kappa \). If the oracle test is successful, the control parameter \( \kappa \) is checked. If the control parameter is true, the dynamic subscript test is made. If this test also succeeds, the value of the expression \( e \) is returned. If the control parameter is false, the propagation stops and the value \( \perp \) is the result. This ordering of the predicate test \( \kappa \) and the dynamic subscript test is appropriate since the predicate may be a guard for the subscript expression.

The correspondence between \( \mathcal{V} \) and \( C \) is shown by the following theorem.

**Theorem 4.2** The semantics \( C \) satisfies the invariant

\[
\mathcal{V}[P]ib\mathcal{O} \subseteq C[P]ib\mathcal{T}\mathcal{O}
\]

for any \( i, b, p \) if the array dependence oracle \( \mathcal{O}_P \) is conservative.

**Proof** The proof proceeds by induction on the structure of the program \( P \). The only interesting case is the array update clause. The applicable

\[
\mathcal{O} \in or = ide_a \rightarrow exp \rightarrow ide_a \rightarrow exp \rightarrow bool
\]

\[
OA : stmt \rightarrow ide \rightarrow val \rightarrow or \rightarrow bool
\]

\[
OA[x := e] = \lambda ib. i \in ide_e \rightarrow (i \equiv x \rightarrow \mathbf{T}, \mathbf{F}), F
\]

\[
OA[a[e'] := e] = \lambda ib. i \in ide_e \rightarrow O(a, e', i, b), F
\]

\[
OA[if e s_t s_f] = \lambda ib. OA[s_t]ib \lor OA[s_f]ib
\]

\[
OA[s_1; s_2] = \lambda ib. OA[s_1]ib \lor OA[s_2]ib
\]

\[
OA[while e s] = \lambda ib. OA[s]ib
\]

\[
OA[end] = \lambda ib.F
\]

**Figure 4.4** Definitions for \( OA \)
denotations are as follows:

\[ \forall[a[e']] := e ib \sigma = \mathcal{O}_P(a, e', i, b) \rightarrow ((a, \mathcal{E}[e'] \sigma) \xrightarrow{?} (i, b) \rightarrow \mathcal{E}[e] \sigma, \sigma ib), \sigma ib \]

and

\[ \mathcal{C}[a[e']] := e ib \kappa \sigma = \mathcal{O}_P(a, e', i, b) \rightarrow (\kappa \rightarrow ((a, \mathcal{E}[e'] \sigma) \xrightarrow{?} (i, b) \rightarrow \mathcal{E}[e] \sigma, \sigma ib), \bot), \sigma ib \]

The result is immediate if the oracle returns \texttt{false}. The rest of the case proceeds as in the proof of language \( \mathcal{W} \) for the assignment clause. \( \Box \)

4.3.2 Program Dependence Trees for Arrays

The next step of the derivation prepares the semantics for the staging derivation by identifying all parameters that are dependent upon the initial store. This transformation proceeds on the array semantics in the same manner as in the core semantics; the new meaning function appears in Figure 4.5.

The invariant for this stage of the derivation is established by the following theorem:

**Theorem 4.3** The semantics \( \mathcal{C}' \) satisfies the invariant

\[ \mathcal{C}[P] ib T \sigma = \mathcal{C}'[P] ib (\lambda \sigma. T)(\lambda ib \sigma. \sigma ib) \sigma \]

for any \( i, b, p \) if the array dependence oracle \( \mathcal{O}_P \) is conservative and deterministic.

**Proof** The proof proceeds in the same fashion as the proof for the core language. The array oracle does not introduce any complications in the step. \( \Box \)

The final step of the derivation separates the function \( \mathcal{C}' \) into its static and dynamic components. The function \( \mathcal{G} \) builds the \( \textit{pdt} \) while the function \( \mathcal{P} \) interprets the \( \textit{pdt} \). Figure 4.6 presents the clauses for the \( \mathcal{G} \) function, with the auxiliary functions, and the \( \mathcal{P} \) and \( \mathcal{P}_k \) functions appearing in Figure 4.7.

There are several differences in this \( \mathcal{G} \) function. While the clauses for the \texttt{if}, \texttt{while}, \texttt{end}, and assignment statements, and statement composition remain essentially the
Semantic Domains:

\[
\begin{align*}
 b, v & \in \text{val}^T \quad = \text{val} \cup \{\top\} \\
x & \in \text{ide}_a \\
a & \in \text{ide}_a \\
i & \in \text{ide} \quad = \text{ide}_a \oplus \text{ide}_s \\
e, e', p & \in \text{exp} \quad \text{(abstract)} \\
\kappa & \in \text{store} \rightarrow \text{bool}^T \\
\gamma & \in \text{ide} \rightarrow \text{store}^T \rightarrow \text{val}^T \\
\sigma & \in \text{store}^T \quad = \text{ide} \rightarrow \text{val}^T \rightarrow \text{val}^T \rightarrow \text{val}^T
\end{align*}
\]

Semantics:

\[
\begin{align*}
\mathcal{E} & : \text{exp} \rightarrow \text{store}^T \rightarrow \text{val}^T \quad \text{(abstract)} \\
\mathcal{C}' & : \text{update} \rightarrow \text{stmt} \rightarrow \text{ide} \rightarrow \text{val}^T \rightarrow (\text{store}^T \rightarrow \text{bool}^T) \\
& \quad \rightarrow (\text{ide} \rightarrow \text{store}^T \rightarrow \text{val}^T) \rightarrow \text{store}^T \rightarrow \text{val}^T
\end{align*}
\]

\[
\begin{align*}
\mathbb{A}_{C'} & : C'[x := e] = \lambda b_i.\gamma.i \vdash x \rightarrow (\lambda \sigma.\kappa \sigma \rightarrow E[e](\lambda j.e.\gamma j\sigma), \bot), \lambda \sigma.\gamma ib\sigma \\
\mathbb{I}_{C'} : C'[\text{if } p s_t s_f] & = \lambda b_i.k \gamma.\neg(\mathcal{O}(s_t, i, b) \lor \mathcal{O}(s_f, i, b)) \rightarrow \lambda \sigma.\gamma ib\sigma, \\
& \text{let } \kappa^+ = \lambda \sigma.\kappa \sigma \land \neg E[p](\lambda j.e.\gamma j\sigma), \\
& \kappa^- = \lambda \sigma.\kappa \sigma \land \neg E[p](\lambda j.e.\gamma j\sigma) \in \\
& (\neg(\mathcal{O}(s_t, i, b) \rightarrow (\lambda \sigma.\kappa^+ \sigma \rightarrow \gamma ib\sigma, \bot), C'[s_t] ibk^+ \gamma) \cup \\
& (\neg(\mathcal{O}(s_f, i, b) \rightarrow (\lambda \sigma.\kappa^- \sigma \rightarrow \gamma ib\sigma, \bot), C'[s_f] ibk^- \gamma) \\
\mathbb{U}_{C'} : C'[a[e']] := e & = \lambda b_i.k \gamma.O(a, e', i, b) \rightarrow (\kappa \sigma \rightarrow \\
& (((a, E[e'](\lambda j.e.\gamma j\sigma)) \vdash (i, b) \rightarrow (\lambda \sigma.\kappa \sigma \rightarrow E[e](\lambda j.e.\gamma j\sigma), \\
& \lambda \sigma.\gamma ib\sigma), \bot), \lambda \sigma.\gamma ib\sigma \\
\mathbb{C}_{C'} & : C'[s_1; s_2] = \lambda b_i.k \gamma.C'[s_2] ibk(\lambda j.e.C'[s_1] jck\gamma) \\
\mathbb{W}_{C'} : C'[\text{while } p s] & = \text{fix}(\lambda w.\lambda b_i.k \gamma.\neg\mathcal{O}(s, i, b) \rightarrow \lambda \sigma.\gamma ib\sigma, \\
& \text{let } \kappa^+ = \lambda \sigma.\kappa \sigma \land E[p](\lambda j.e.\gamma j\sigma), \\
& \kappa^- = \lambda \sigma.\kappa \sigma \land E[p](\lambda j.e.\gamma j\sigma) \in \\
& (\lambda \sigma.\kappa^- \sigma \rightarrow \gamma ib\sigma, \bot) \cup \text{wibk}^+(\lambda j.e.C'[s] jck^+ \gamma)) \\
\mathbb{F}_{C'} & : C'[s; \text{end}] = \lambda b_i.k \gamma.C'[s] ibk\gamma
\end{align*}
\]

**Figure 4.5 Abstraction Stage of Array Semantics**
Semantic Domains:

$$v \in val^T = val \cup \{\top\} \text{ (abstract)}$$
$$x \in ide_s$$
$$a \in ide_a$$
$$i \in ide = ide_a \oplus ide_s$$
$$e, e', p \in exp \text{ (abstract)}$$
$$f \in dem = \mathcal{P}_{fin}(exp)$$
$$\mathcal{O} \in or = ide_a \rightarrow exp \rightarrow ide_a \rightarrow exp \rightarrow bool$$
$$\sigma \in store^T = ide \rightarrow val^T \rightarrow val^T \rightarrow val^T$$
$$c_j \in code = d-node \oplus (d-node \otimes d-node) \oplus d-node^+$$
$$\gamma, c_c \in code-tbl = ide \rightarrow code$$
$$c_d \in d-node = (p-node \otimes exp \otimes code-tbl) \oplus (p-node \otimes ide) \oplus$$
$$(p-node \otimes code) \oplus (ide \otimes code) \oplus$$
$$(p-node \otimes exp \otimes code-tbl \otimes exp \otimes code-tbl \otimes code) \oplus$$
$$(p-node \otimes exp \otimes code-tbl \otimes exp \otimes code-tbl) \oplus$$
$$\kappa, c_k \in p-node = \text{true} \oplus (p-node \otimes exp \otimes code-tbl) \oplus$$
$$(p-node \otimes exp \otimes code-tbl)$$

Semantics:

$$\mathcal{G} : \text{stmt} \rightarrow ide \rightarrow dem \rightarrow p-node \rightarrow code-tbl \rightarrow code$$
$$\mathcal{E} : exp \rightarrow store^T \rightarrow val^T \text{ (abstract)}$$
$$\mathcal{P} : code \rightarrow store^T \rightarrow val^T \rightarrow val^T$$
$$\mathcal{P}_k : p-node \rightarrow store^T \rightarrow bool^T$$

$$\mathcal{G}[x := e] = \lambda i f \kappa \gamma. i \downarrow x \rightarrow (\kappa, e, \mathcal{M}(e, \gamma), \gamma if)$$
$$\mathcal{G}[\text{if } p \ s_1 \ s_2] = \lambda i f \kappa \gamma. (\mathcal{O}A(s_1, i, f) \lor \mathcal{O}A(s_2, i, f)) \rightarrow \gamma if,$$
\hspace{1cm} let \(\kappa^+ = (T, \kappa, p, \mathcal{M}(p, \gamma)), \kappa^- = (F, \kappa, p, \mathcal{M}(p, \gamma))\) in
\hspace{1cm} \(((\neg \mathcal{O}A(s_1, i, f) \rightarrow \mathcal{M}V(\kappa^+, i, s, \gamma), \mathcal{G}[i f \kappa^+ \gamma]),$$
\hspace{1cm} \((\neg \mathcal{O}A(s_2, i, f) \rightarrow \mathcal{M}V(\kappa^-, i, s, \gamma), \mathcal{G}[f f \kappa^- \gamma]))$$

$$\mathcal{G}[a[e'] := e] = \lambda i f \kappa \gamma. \mathcal{O}S(a, e', i, f) \rightarrow (\kappa, e', \mathcal{M}(e', \gamma), e, \mathcal{M}(e, \gamma), \gamma if), \gamma if$$
$$\mathcal{G}[s_1; s_2] = \lambda i f \kappa \gamma. \mathcal{G}[s_2] i f \kappa(\lambda j c. \mathcal{G}[s_1] j c \kappa)$$
$$\mathcal{G}[\text{while } p \ s] = \text{fix}(\lambda w. \lambda i f \kappa \gamma. \neg \mathcal{O}A(s, i, f) \rightarrow \gamma if,$$
\hspace{1cm} let \(\kappa^+ = (T, \kappa, p, \mathcal{M}(p, \gamma)), \kappa^- = (F, \kappa, p, \mathcal{M}(p, \gamma))\) in
\hspace{1cm} \(\mathcal{M}V(\kappa^-, i, f, \gamma) \circ w if \kappa^+(\lambda j c. \mathcal{G}[s] j c \kappa^+ \gamma)$$

$$\mathcal{G}[s; \text{end}] = \lambda i f \kappa \gamma. (i, \mathcal{G}[s] i f \kappa \gamma)$$

Figure 4.6  Merge Node Semantics for Arrays
\[
\begin{align*}
\mathcal{P}[(\kappa, e, C_e)] &= \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \rightarrow \lambda t. E[e] RS(C_e, \sigma), \lambda t. \bot \\
\mathcal{P}[(C_i, C_f)] &= \lambda \sigma. \mathcal{P}[C_i] \sigma \cup \mathcal{P}[C_f] \sigma \\
\mathcal{P}[[C_1, C_2, \ldots]] &= \lambda \sigma. \mathcal{P}[C_i] \sigma \cup \mathcal{P}[[C_2, \ldots]] \sigma \\
\mathcal{P}[(\kappa, C)] &= \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \rightarrow \mathcal{P}[C] \sigma, \lambda t. \bot \\
\mathcal{P}[(\kappa, i)] &= \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \rightarrow \lambda t. \sigma i t, \lambda t. \bot \\
\mathcal{P}[(i, C_i)] &= \lambda \sigma. \mathcal{P}[C_i] \sigma \\
\mathcal{P}[(\kappa, e', C'_e, e, C_e, C_r)] &= \lambda \sigma. \mathcal{P}_k[\kappa] \sigma \rightarrow J_S(\mathcal{P}[(\kappa, e', C'_e, e, C_e)] \sigma, \mathcal{P}[C_r] \sigma), \lambda t. \bot \\
\mathcal{P}[(\kappa, e', C'_e, e, C_e)] &= \lambda \sigma. \lambda t. t = E[e'] RS(C'_e, \sigma) \rightarrow E[e] RS(C_e, \sigma), \bot \\
\mathcal{P}_k[true_e] &= \lambda \sigma. T \\
\mathcal{P}_k[(T, \kappa, p, C_p)] &= \lambda \sigma. E[p] RS(C_p, \sigma) \land \mathcal{P}_k[\kappa] \sigma \\
\mathcal{P}_k[(F, \kappa, p, C_p)] &= \lambda \sigma. \neg E[p] RS(C_p, \sigma) \land \mathcal{P}_k[\kappa] \sigma \\
\mathcal{M}_S(e, \gamma) &= \{(j, \gamma j 0) \mid j \in R(e) \cap ide_a\} \cup \\
& \{(a, \gamma a \{j \mid j \in AR(e, a)\}) \mid a \in R(e) \cap ide_a\} \\
\mathcal{M}_V(\kappa, i, b, \gamma) &= (\kappa, \gamma i b) \\
\mathcal{R}_S(C_e, \sigma) &= \{(j, \lambda t.(\mathcal{P}[C_j] \sigma) t) \mid (j, C_j) \in C_e\} \\
\mathcal{O}_A[s] &= \lambda i f O. \exists b \in f. OA(s, i, b, O) \\
\mathcal{Y}[(\kappa, e', C'_e, e, C_e, C_r)] &= \lambda \sigma t. \mathcal{P}_k[\kappa] \sigma \rightarrow (t = E[e'] RS(C'_e, \sigma) \rightarrow T, \mathcal{Y}(C_r, \sigma, t)), \mathcal{Y}(C_r, \sigma, t) \\
\mathcal{Y}[(\kappa, e', C'_e, e, C_e)] &= \lambda \sigma t. \mathcal{P}_k[\kappa] \sigma \rightarrow (t = E[e'] RS(C'_e, \sigma) \rightarrow T, F), F
\end{align*}
\]

**Figure 4.7** Help Functions for Array Semantics
same, the auxiliary function $\mathcal{MV}$ now accommodates array identifiers as well as simple identifiers. The function that creates the valve nodes,

$$\mathcal{MV}(\kappa, i, b, \gamma) = (\kappa, \gamma ib)$$

now makes a separate node type instead of using identity assignment statements for valve nodes. In the case of simple identifiers, an identity assignment is sufficient. However, for arrays, only the index expression for the demand is known when then valve node is created. An identity assignment would require the information to compute value of the index expression to be propagated along with the demand, since the value of the index expression at the point that the valve node is created does not necessarily equal the value of that expression at the point in the program that generated the demand. Using explicit valve nodes for arrays prevents this propagation and simplifies the semantics.

Demands for array identifiers are propagated differently in $\mathcal{G}$ than in the previous stages. In the earlier stages, the demanded index is always a value, since the propagation happens either through expression evaluation or from the initial demand. In $\mathcal{G}$, however, the demands that are propagated for individual array elements are expressions. The merge nodes in the code-tables are labeled with index expressions as opposed to index values. When the store is built to evaluate expressions in the $\mathcal{P}$ function, a mechanism must exist to select the appropriate code-table entry. Therefore, the index propagated in the $\mathcal{G}$ function is a set of index expressions including all of the index expressions for the given array required to evaluate the expression. This set is finite since expressions are finite. The $\mathcal{MS}$ function collects the code-tables from $\gamma$ required for expression $e$:

$$\mathcal{MS}(e, \gamma) = \{(j, \gamma j 0) | j \in \mathcal{R}(e) \cap ide_a\} \cup \{(a, \gamma a\{j | j \in \mathcal{AR}(e, a)\}) | a \in \mathcal{R}(e) \cap ide_a\}$$

The function $\mathcal{AR}(e, a)$ selects the index expressions for array $a$ referenced in expression $e$. For simple identifiers referenced in expression $e$, demand is propagated as before; for array identifiers, the index argument propagated is a set containing all index expressions for array $a$ in expression $e$.

The array update clause,

$$\mathcal{G}[a[e] := e] = \lambda \kappa f \gamma. \mathcal{O}_{\mathcal{S}}(a, e', i, f) \rightarrow (\kappa, e', \mathcal{MS}(e', \gamma), e, \mathcal{MS}(e, \gamma), \gamma ib), \gamma if$$
performs the oracle test, using the function $\text{OS}$, for each of the index expressions in the demand set $f$. If the expression $e'$ may satisfy any of the demands in the demand set $f$, an array update node is included in the code-table for this statement. The demand is passed further on to collect other statements that may satisfy the demand. Since the run-time store is not available, demand propagation does not stop with a positive answer from the oracle.

The array update clause creates a new node type called an array merge node that represents the partial array mentioned earlier in this chapter. As the demand for an array index is propagated, any update operation that may be required to satisfy the demand is included as a component of this merge node, assembling the section of the array that can not be excluded by the static subscript analysis encoded in the oracle.

These array merge nodes are similar to merge nodes [37] for simple identifiers. Merge nodes for simple identifiers collect assignments for the same identifier from different control paths and pass the value on to uses of the identifier. The array merge node collects all the possible assignments that may satisfy a demand from the different control flow paths to the demand. Since the subscript analysis is imprecise, the array merge node may include update statements that are not actually required to satisfy the demand. The partial array represented by this array merge node is the structure used for each array access. We use the term merge node to refer to an array merge node, unless otherwise specified.

Each merge node contains the information from one update statement along with the code needed to compute the rest of the partial array required that may be required to satisfy the demand. The node includes the index expression from the update statement and the code-table required to compute the expression, the value expression and its code-table, the control node for this path in the program, and the code to compute the other elements that might satisfy the demand.

The $\mathcal{P}$ function, given in Figure 4.6, contains clauses to interpret the pdt nodes. The meaning of an individual array node is a function returning ⊥ if the run-time index does not match that of the update node, and returning the expression value otherwise:

$$\mathcal{P}[\langle \kappa, e', C_e', e, C_e \rangle] = \lambda \sigma. \lambda t. t \overset{\circ}{=} E[e'] \mathcal{RS}(C_e', \sigma) \rightarrow E[e] \mathcal{RS}(C_e, \sigma), \bot$$

The merge node clause first checks the control node to see if this node is enabled. If not, ⊥ is the result, regardless of the run-time value. Otherwise, the portion of the
partial array specified by the first component of the merge node is joined with the partial array specified by the second component of the merge node, using the function $J_S$:

$$\mathcal{P}[[\langle \kappa, e', C_\sigma, e, C_e, C_r \rangle]] = \lambda \sigma. J_S(\mathcal{P}[[\langle \kappa, e', C_\sigma, e, C_e \rangle]], \mathcal{P}[C_r], \sigma)$$

This join function constructs a partial array from pairs of partial arrays. Each partial array respects the ordering of the array update statements in the program; the subscript analysis oracle prunes updates from the program, but since the analysis is imprecise, the ordering information is still required to ensure the correct value results. A partial array $m$ is a function $m : val^T \rightarrow val^T$ from index values to expression values. The function $J_S$ is defined as follows:

$$J_S(m_1, m_2, \sigma) = \lambda t. Y(m_1, \sigma, t) \rightarrow m_2 t, m_1 t$$

where $Y(m, \sigma, t)$ is true if some array update node in $m$ applies to index value $t$ under store $\sigma$. The definition for $Y$ appears in Figure 4.7. Thus, partial arrays are checked, left to right, until the first update is discovered for the demanded index value $t$.

With the addition of array nodes, the type of the $\mathcal{P}$ function includes another parameter, the index value for the demand. The index demand propagated in the $G$ function is an expression. With the initial store available, the value of these index expressions can be computed. This value is used specifically in the array nodes to select the appropriate entry in the array. The value is ignored by the other nodes. Thus, the clauses for the assignment and idf nodes include changes to accept this additional argument. In addition, there is a clause for value nodes,

$$\mathcal{P}[[\langle \kappa, C \rangle]] = \lambda \sigma. P_k[\kappa] \sigma \rightarrow \mathcal{P}[C] \sigma, \lambda t. \bot$$

If the control node is true, the meaning of the code-table is the meaning of the value node. Otherwise, the value is $\bot$ regardless of the value of the index value.

Because all nodes are functions of the index value, the $\mathcal{RS}$ function is changed to apply the result of interpreting the node to the index value:

$$\mathcal{RS}(C_e, \sigma) = \{(j, (\mathcal{P}[[C_j] \sigma])) \mid (j, C_j) \in C_e\}$$

creates a store from the code-table component for a node. Each pair, $(j, C_j)$ in the code-table corresponds to an entry in the finite store for the expression. Stores are functions from identifiers and values to values, and the function in the store for identifier $j$ applies the function found by $\mathcal{P}$ from the code $C_j$ in the code table to the index value.
The functions $C'$, $P$ and $G$ satisfy an invariant similar to that defined for the core language functions:

$$P(G\bar{P}[i\{b\}(true_c)\gamma_0]\bar{s}) = C'\bar{P}[ib(\lambda\sigma.T)(\lambda i\sigma.\sigma)i]\bar{s}$$

with the code tree parameter $\gamma_0$ defined as:

$$\lambda is.i \in ide_s \rightarrow (\text{true}_c, i), (\mathcal{OS}(i, s) \rightarrow (\text{true}_c, i), \langle \rangle)$$

This initial code tree parameter creates the $\text{idef}$ nodes for simple identifiers and array identifiers if the demanded index is not assigned before being used. The oracle is used to ensure that no unnecessary $\text{idef}$ nodes are created. This test is necessary since demand propagation continues for array identifiers throughout the program since a positive oracle answer does not guarantee that the demanded index is assigned in an update statement. The following theorem proves this correspondence.

**Theorem 4.4** The functions $C'$, $P$ and $G$ satisfy the following invariant for any identifier $i$, index value $b$, and program $P$:

$$P(G\bar{P}[i\{b\}(true_c)\gamma_0]\bar{s}) = C'\bar{P}[ib(\lambda\sigma.T)(\lambda i\sigma.\sigma)i]\bar{s}$$

if the array dependence oracle $\mathcal{O}_P$ is conservative, and the code tree parameter $\gamma_0$ is defined as above.

**Proof** The proof requires the lemma below. The theorem follows directly from the lemma.

**Lemma 4.1** For program $P$, identifier $i$, index expression set $f$, index value $\bar{b}$ where $\bar{b}$ is the value of some expression $b \in f$, such that

$$\mathcal{O}(a, e', i, b) = \mathcal{O}(a, e', i, \bar{b})$$

for all oracle queries in program $P$, store $\sigma$, parameterized store $\bar{\gamma}$, and code-table $\gamma$ such that

$$\bar{\gamma}ib\sigma = (P[\gamma i]\sigma)\bar{b}$$

for all identifiers $i$, and control parameter $\kappa$ and $\bar{k}$ such that

$$\bar{k}\sigma = P_k[\kappa]\sigma$$

then

$$(P[\mathcal{G}[P][ibk\gamma]\sigma)\bar{b} = C'[P][ib\bar{k}\bar{\gamma}\sigma]$$
Proof The proof proceeds by induction on the structure of $P$. The array update and valve node clauses are the only interesting cases.

The applicable denotations for array updates are as follows:

$$C'[a[e'] := e][ibk\gamma\sigma] = O(a, e', i, \overline{b}) \rightarrow (\kappa \sigma \rightarrow ((a, \mathcal{E}[e']((\lambda j.c.\gamma jc\sigma))) \overset{?}{=} (i, \overline{b}) \rightarrow (\lambda\sigma.\kappa\sigma \rightarrow \mathcal{E}[e)((\lambda j.c.\gamma jc\sigma), \lambda\sigma.\gamma i\overline{b}\sigma)), \bot), \lambda\sigma.\gamma i\overline{b}\sigma)$$

and

$$(\mathcal{P}[\mathcal{G}[a[e'] := e][if\kappa\gamma\sigma])\overline{b} = (O\mathcal{A}'(a, e', i, f) \rightarrow \mathcal{P}[\kappa, e', MS(e', \gamma), e, MS(e, \gamma), \gamma if]\sigma, \mathcal{P}[\gamma is]\sigma)\overline{b}$$

By the condition on $b$ and $\overline{b}$, if the oracle returns false for $b$, the oracle also returns false for $\overline{b}$. By the assumptions on the input stores, the result is immediate in this case.

If $\overline{b} = \bot$, the result holds immediately since the expression $b \in f$ appears in the demand set for some array, and this expression computation will diverge based on the assumptions on the inputs.

If the oracle returns true, the denotations are as follows:

$$C'[a[e'] := e][ibk\gamma\sigma] = \kappa \sigma \rightarrow ((a, \mathcal{E}[e']((\lambda j.c.\gamma jc\sigma))) \overset{?}{=} (i, \overline{b}) \rightarrow (\lambda\sigma.\kappa\sigma \rightarrow \mathcal{E}[e][(\lambda j.c.\gamma jc\sigma), \lambda\sigma.\gamma i\overline{b}\sigma]), \bot)$$

and

$$(\mathcal{P}[\mathcal{G}[a[e'] := e][if\kappa\gamma\sigma])\overline{b} = (\mathcal{P}[\kappa, e', MS(e', \gamma), e, MS(e, \gamma), \gamma if]\sigma, \mathcal{P}[C_r]\sigma)$

By the assumptions on the input control parameters, $\overline{b}\sigma$ is false if and only if $\mathcal{P}_k[\kappa]\sigma$ is false. In this case, the result is immediate.

If the control parameters return true, the denotations are as follows:

$$C'[a[e'] := e][ibk\gamma\sigma] = (a, \mathcal{E}[e']((\lambda j.c.\gamma jc\sigma))) \overset{?}{=} (i, \overline{b}) \rightarrow (\lambda\sigma.\kappa\sigma \rightarrow \mathcal{E}[e][(\lambda j.c.\gamma jc\sigma), \lambda\sigma.\gamma i\overline{b}\sigma)$$
and
\[
(P[G[a[e] := e][i \kappa \gamma][\sigma])\tilde{s} = (JS(\langle E[e']RS(C_e', \sigma), E[e]RS(C_e, \sigma)\rangle), P[C_r][\sigma])\tilde{b}
\]

These denotations are the same based on the assumptions on the input parameters and the definition of the function \(JS\).

The other part of the proof that differs in the array case is the implementation of valve nodes. In the previous chapter, valve nodes were implemented using identity assignment nodes. With the addition of arrays, a distinct node type is used for valve nodes. The denotation for valve nodes from \(G\) and \(P\) as well as from the \(C'\) function are as follows:

\[
C' : \quad \kappa^+\sigma \rightarrow \gamma i\bar{b}\sigma, \bot
\]
\[
P : \quad (P(\langle MV(\kappa^+, i, f, \gamma)\rangle)\tilde{b}
= (P(\langle \kappa, \gamma if \rangle)\sigma)\tilde{b}
= (P_\kappa[\kappa][\sigma \rightarrow P[\gamma if][\sigma, \lambda t. \bot])]\tilde{b}
\]

By the assumptions on the input arguments, these denotations are equal, and the theorem holds.

The other cases follow as in the previous chapter since the run-time index is ignored in the rest of the cases.

\[\Box\]

4.3.3 PDT Example with Arrays

The following example program,

\[
\begin{align*}
\text{if } p_1 & \quad a[i] := 6 & a[i] := 5; \\
\text{if } p_2 & \quad a[j] := 4 & a[k] := 4;
\end{align*}
\]

end

demonstrates the creation of the \texttt{merge} node. With an initial demand for array \(a\) at index 1, and if the subscript analysis determines that \(k\) can not be 1, the \texttt{pdt} in Figure 4.8 is created. The figure does not include the nodes representing the demand to the initial store and the flow edges for the computation of the subscripts and the predicates. In addition, the nodes are not duplicated. The assignment to \(a[j]\) is the first component of a \texttt{merge} node. The edges labeled \(S\) to the merge node represent the left-to-right ordering within the merge nodes. If the update for index \(j\) does
Figure 4.8 Array Merge Node Example
not satisfy the demand, the nodes representing the second partial array in the join operation, which is the pair of \(a[i]\) updates, supply the value. The demand propagates over the \(S\) edges until an array update is found that contains the demanded index. The sequencing provided by the \(S\) edges is required in case \(i = j = 1\). The picture also demonstrates that the array valve nodes perform the same function as their counterparts for simple identifiers.

### 4.4 Operational Semantics for Array PDTs

The derivation in the previous section adds two new node sets to the \(pdt\) graph: valve nodes and merge nodes. The new specification for \(pdt\) graph \(G\) is as follows:

\[
G = (N_I, N_S, N_V, N_M, N_P, N_F, S)
\]

with \(N_V\) representing the valve node set and \(N_M\) the merge node set. All other sets remain the same. The mapping from the tree to the graph presentation follows the same strategy as that for the core language. Like if nodes, merge nodes have two kinds of successors: flow successors and array successors. Since the partial arrays are treated as values, flow successors for merge nodes are treated the same as flow successors for assignment nodes. For array successors, however, the merge node is providing an incomplete version of the partial array to which the successor merge node will add its contribution. Thus, the propagation of this value is not handled with the substitution function. A tag is added to the merge node, similar to the true/false tag for the if node, designating which type of successor the merge node has. As with the assignment nodes for simple identifiers, the label of the node includes the expression and the identity of the target of the assignment. The label for merge nodes also includes a component to receive the partial array to be augmented. Figure 4.9 gives the node specifications.

The rewriting rules for the other nodes remain the essentially the same, as does the definition for substitution. The rewriting rules for all node sets appear in Figure 4.10.

The rewriting rules for idef and assignment nodes propagate their value to nodes in the valve and merge node sets in addition to the other sets. The valve node clause is similar to the idef and assignment node clauses except that the redex is removed from the valve node set. In addition, the expression portion of the valve node is not evaluated by the expression evaluator since the expression will be a value, not an expression, when the valve node becomes a redex.
\[
\begin{align*}
\text{idf} &= \langle i_m, x \rangle \\
\text{set} &= \langle s_m, x := e \rangle \\
\text{array} &= \langle a_{m}, a[e'] := e, p \rangle \\
\text{merge} &= \langle a'_{m}, a[e'] := e, p \rangle \\
\text{start} &= \langle p_{m}^{\star}, \text{true}_e \rangle \\
\text{true} &= \langle p_{m}^{\star}, e \rangle \\
\text{false} &= \langle p_{m}^{\star}, e \rangle \\
\text{valve} &= \langle v_m, x := e \rangle \\
\text{final} &= \langle f_m, x \rangle 
\end{align*}
\]

**Figure 4.9** Representations for PDT Nodes

The rewriting rule for the **merge** node with an array successor, rule \( M_m \), builds a new partial array and updates the partial array portion of the successor with the new partial array. The **merge** node with a flow successor, using rule \( M_f \), propagates the new partial array into the expression of the successor node just like a simple value. The expression evaluator uses function application to extract the correct value from the partial array when an array access expression is evaluated.

The definition for both the sequential and parallel rewriting relations extend to the array rewriting rules. However, since the comparison of function values is undecidable and the values being propagated are functions from index values to values, the definition of the conflict predicate \( C \) is as follows:

\[
C(q, x, s) = \exists j, k \in s. (id(j) = id(k) = x) \land (S(j) = S(k) = q)
\]

for identifier \( x \), node \( q \) and redex set \( s \).

### 4.4.1 Rewriting Example

To clarify the function of the array rewriting rule, the example from the previous section is rewritten in this section. Figure 4.11 reproduces the *pdt graph* from that section. Once again, to simplify the pictures, demands to the initial store and the computation of array index expressions are ignored, and nodes are not duplicated in the figure. For the first 2 rewriting steps, the predicate nodes \( p1 \) and \( p2 \) are rewritten, assuming the both predicates are **true**. Figure 4.12 contains the result of these steps. At this point, the **merge** nodes and the demand node remain. The **merge** node for
\[
\begin{align*}
\text{I: } & \langle i_r, x \rangle, S(r) = n \quad \begin{cases}
N'_I &= N_I - \{r\} \\
N'_k &= N_k[n \leftarrow x/\sigma x] \text{ for } k \in \{S, V, M, P, F\}
\end{cases} \\
\text{S: } & \langle s_r, x := e \rangle, S(r) = n \quad \begin{cases}
N'_I &= N_I \\
N'_S &= (N_S - \{r\})[n \leftarrow x/E[e]] \\
N'_k &= N_k[n \leftarrow x/E[e]] \text{ for } k \in \{V, M, P, F\}
\end{cases} \\
\text{V: } & \langle v_r, x := e \rangle, S(r) = n \quad \begin{cases}
N'_I &= N_I \\
N'_k &= N_k[n \leftarrow x/e] \text{ for } k \in \{S, M, P, F\} \\
N'_V &= (N_V - \{r\})[n \leftarrow x/e]
\end{cases} \\
\text{M}_f: \ & \langle a_f, a[e'] := e, p \rangle, S(r) = n \quad \begin{cases}
N'_I &= N_I \\
N'_k &= N_k[n \leftarrow a / J(p, E[e'], E[e])] \\
N'_M &= (N_M - \{r\})[n \leftarrow a / J(p, E[e'], E[e])]
\end{cases} \text{ for } k \in \{S, V, P, F\} \\
\text{M}_m: \ & \langle a_m, a[e'] := e, p \rangle, S(r) = n \quad \begin{cases}
N'_k &= N_k \text{ for } k \in \{I, S, V, P, F\} \\
N'_M &= N_M - \{(a_n, a'[e_1] := e', p')\} \cup \{(a_n, a'[e_1] := e', J(p, E[e'], E[e']))\}
\end{cases} \\
\text{P: } & \langle p'_m, e, n \rangle \quad \begin{cases}
N'_k &= N_k - Q_k \text{ for } k \in \{I, S, V, M, P, F\} \\
N'_p &= N_P - Q_P - \{r\}
\end{cases}
\end{align*}
\]

for \( b \in \{t, f, s\} \)

where

\[
Q = \begin{cases}
\emptyset & \text{if } E[e] = b \lor e \equiv \text{true}_c \\
\{m \mid \langle p'_m, e \rangle, S(q) = m \in Q\} \cup \{n\} & \text{if } E[e] \neq b \\
\end{cases}
\]

\[
Q_i = \{m \mid m \in Q \land m \in N_i\} \text{ for } i \in \{I, S, V, M, P, F\}
\]

**Figure 4.10** Rewriting Rules for the Array PDTS
Figure 4.11  Array Rewriting Example - Original Program

Figure 4.12  Array Rewriting Example - Predicates Removed
$a[i]$ is the only redex in the graph, so the next step rewrites that node. We assume that $i = 1$. The partial array $1 \rightarrow 6$ is sent to the merge node for $a[j]$, reflecting the other computations for the demand. Figure 4.13 contains the resulting graph. The final step rewrites the merge node for $a[j]$. We assume that $j = 1$. The join operation overrides the mapping $1 \rightarrow 6$ with the new information from the update for $j$. Thus, the correct value for $a[1]$ propagates to the final demand.

This chapter examined the effects on the $pdt$ which result from the addition of array update and array access operations. The $pdt$ for programs using arrays include merge nodes which represent the portion of the array that may be required to fulfill an array demand. These merge nodes are a compromise between the functional approach to array values presented in the previous chapter and the shared memory approach for arrays generally used in the compiling literature. With the partial arrays denoted by the merge nodes, the data flow patterns in a program can be examined, independent of the storage-access dependences that arise in a shared memory implementation of arrays. The additional storage dependences are derivable from the information in the $pdt$; thus, the $pdt$ contains enough information to be useful in a shared memory setting.

---

**Figure 4.13** Array Rewriting Example - First Merge Node
Chapter 5

Semantics of Dependence for Abort

Although structured programs provide a simple framework in which to understand dependence, most programming languages include some form of non-local control construct in the language definition. These non-local constructs complicate the control dependences that are possible within a program. We begin to explore these more complex control dependence relations by adding an abort construct to our structured language. To simplify the presentation, abort is added to the core language instead of the language with arrays. The abort construct, transferring control to the end of the program from anywhere in the program, is the simplest non-structured control construct. Although the control dependence relation does not change as radically as with the introduction of general control in the next chapter, the relation no longer only reflects the nesting structure of the program.

Section 5.1 describes the abort statement and its semantics. Section 5.2 and Section 5.5 present two denotational perspectives on control dependences in the presence of an abort construct. Section 5.6 describes the impact on the control dependence relation caused by the addition of the abort construct to the language. The operational semantics for pdt graphs in the presence of the abort dependences appears in Section 5.7. The two approaches to describing the abort construct are compared in the final section.

5.1 Language Semantics for Abort

The abort statement is added to the syntax of the core language as follows:

\[
\begin{align*}
  x & \in \text{ide} \\
  e & \in \text{exp} \\
  s & ::= \text{if } e \text{ s s while } e \text{ s } x := e \text{ s s abort} \\
  P & ::= s ; \text{end}
\end{align*}
\]

Traditionally, semantics for languages with non-structured control constructs have been continuation semantics [54] as opposed to direct semantics such as those pre-
ented so far. Extensions to the direct approach [27] provide a method to define the semantics for the `abort` construct. A direct semantics for `abort` appears in Figure 5.1. In this semantics, the store domain is the disjoint union of standard stores and the abort store, designated with the symbol ☐, which is assumed to be distinguishable from all stores and values. The `abort` clause, E, returns the abort store value ☐. Each clause checks the store it receives to determine if an abort has occurred. The abort store is propagated further if there was an abort; processing proceeds normally otherwise. For example, the assignment clause, A, determines if the store is the abort store; if so, the store is returned unchanged. Otherwise, the result is the updated store. This propagation of the abort store assigns a sequential semantics to the `abort` statement since any `abort` statement that is evaluated alters the meaning of all subsequent statements. The other clauses similarly test for the abort store. The `while` clause checks for the abort store before entering the loop and before each subsequent iteration. Thus, loop execution does not begin if an `abort`
occurred before the loop, and a loop is terminated when an abort occurs within the loop body.

5.2 Denotational Semantics for Abort

The staging analysis begins as before with the denotational definition of a language with abort, presented in Figure 5.1. Next, the identifier parameter from the result store is moved forward in the parameter list from the output store parameter to convert to the demand perspective. Figure 5.2 presents the demand semantics for abort. The abort clause, $E_v$, still returns $\emptyset$, the abort value. In this case, however, the entire store is not an abort value; only the value for the identifier $i$ is affected. As before, the assignment clause, $A_v$, is expanded with the lazy store update function. The predicate $T(e, \sigma)$ is true if any of the values from the store $\sigma$ required to evaluate expression $e$ are abort values. If this predicate is true, the abort value is the result

---

Semantic Domains:

\[
\begin{align*}
  w & \in \text{val} \quad \text{(abstract)} \\
  v & \in \text{aval} = \text{val} + \emptyset \\
  i, x & \in \text{ide} \\
  e, p & \in \text{exp} \quad \text{(abstract)} \\
  \sigma & \in \text{astore} = \text{ide} \rightarrow \text{aval}
\end{align*}
\]

Semantics:

\[
\begin{align*}
  \mathcal{E} & : \text{exp} \rightarrow \text{astore} \rightarrow \text{val} \quad \text{(abstract)} \\
  \mathcal{V} & : \text{stmt} \rightarrow \text{ide} \rightarrow \text{astore} \rightarrow \text{aval}
\end{align*}
\]

\[
\begin{align*}
  A_v : \mathcal{V}[x := e] &= \lambda i.\sigma. i \vDash x \rightarrow (T(e, \sigma) \rightarrow \emptyset, \mathcal{E}[e] \sigma, \sigma i) \\
  I_v : \mathcal{V}[\text{if } p \ s_1 \ s_2] &= \lambda i.\sigma. T(p, \sigma) \rightarrow \emptyset, (\mathcal{E}[p] \sigma \rightarrow V[s_1] i\sigma, V[s_2] i\sigma) \\
  E_v : \mathcal{V}[\text{abort}] &= \lambda i.\sigma. \emptyset \\
  C_v : \mathcal{V}[s_1; s_2] &= \lambda i.\sigma. V[s_2] i\sigma((\lambda j. V[s_1] j\sigma) \\
  W_v : \mathcal{V}[\text{while } p \ s] &= \lambda i.\sigma. \text{fix}(\lambda w. \lambda i.\sigma. T(p, \sigma) \rightarrow \emptyset, \\
  & \quad (\mathcal{E}[p] \sigma \rightarrow V[s] i\sigma \vDash \emptyset \rightarrow \emptyset, w i((\lambda j. V[\emptyset] j\sigma)), \sigma i)) \\
  F_v : \mathcal{V}[s; \text{end}] &= \lambda i.\sigma. V[s] i\sigma
\end{align*}
\]

where

\[
T(e, \sigma) = \emptyset \in \{v \mid v = \sigma k, k \in R(e)\}
\]

\[\textbf{Figure 5.2} \quad \text{Data Stage for Abort Semantics}\]
for the current demand. This test only has an effect if the demanded identifier $i$
matches the identifier $x$ being assigned. Thus, we have altered the semantics of abort
to disregard abort values not arising from demands. We call this interpretation of abort
the \textit{lazy} interpretation.

Since the semantics of the abort statement is altered in the demand semantics,
the abort demand semantics and the standard semantics potentially assign different
meanings to programs. If the demand semantics returns an abort value, then the
standard semantics results in the abort store. If the standard semantics doesn’t
return an abort value, the meanings returned by the two semantics are the same.
The appropriate invariant for the demand semantics is presented in the following
theorem.

\textbf{Theorem 5.1} \ For all programs $P$, stores $\bar{\sigma}$ and extended stores $\sigma$ such
that
\[ (\exists j. j = \emptyset \supset \sigma = \emptyset) \land (\sigma \neq \emptyset \supset (\forall i. \bar{\sigma}_i = \sigma_i)) \]
and all identifiers $i$, the following two conditions hold:

1) $\mathcal{V}[P][i\bar{\sigma} = \emptyset \supset \mathcal{W}[P][\sigma = \emptyset$

2) $\mathcal{W}[P][\sigma \neq \emptyset \supset \mathcal{V}[P][i\bar{\sigma} = \mathcal{W}[P][\sigma_i$

\textbf{Note:} The condition placed on the two input stores $\sigma$ and $\bar{\sigma}$ reflects the
change in the range of the meaning function from $\mathcal{W}$ to $\mathcal{V}$. The meaning
of a program that aborts under the $\mathcal{W}$ function is an abort store while
the meaning under the $\mathcal{V}$ function is a store that contains an abort value
for some identifier. The first conjunct in the store condition states that, if
the input store to the $\mathcal{V}$ function contains any abort value, the input store
to the $\mathcal{W}$ function is the abort store. The second conjunct specifies that,
if the input store for the $\mathcal{W}$ function is not the abort store, the values
in both stores are equal for all identifiers. The theorem demonstrates the
change that occurs in the meaning of abort with the demand perspective.
Part 1 of the theorem states that if $\mathcal{V}$ aborts for some identifier, then the
result of the program under $\mathcal{W}$ is the abort store. Part 2 of the theorem
applies when the program does not abort under the standard semantics.
In this case, the result for any identifier $i$ from $\mathcal{V}$ corresponds to the value
found for $i$ in the result store from $\mathcal{W}$. $\Box_{Note}$
Proof. The proof proceeds by induction on the structure of the program $P$.

Case $x := e$. The denotations for $P$ using $\mathcal{W}$ and $\mathcal{V}$ are as follows:

$$\mathcal{W}[x := e] \sigma = \sigma \overset{2}{=} \emptyset \rightarrow \sigma, \delta_Z(\sigma, x, \mathcal{E}[e] \sigma)$$

$$= \sigma \overset{2}{=} \emptyset \rightarrow \sigma, (i \overset{2}{=} x \rightarrow \delta(\sigma, x, \mathcal{E}[e] \sigma), \sigma)$$

$$\mathcal{V}[x := e] i \bar{\sigma} = i \overset{2}{=} x \rightarrow (T(e, \bar{\sigma}) \rightarrow \emptyset, \mathcal{E}[e] \bar{\sigma}), \bar{\sigma}i$$

For the first part of the theorem, assume

$$\mathcal{V}[x := e] i \bar{\sigma} = \emptyset$$

Clearly, $T(e, \bar{\sigma}) \neq \bot$ since the result is not $\bot$. There are two cases to consider here. If $i \overset{2}{=} x$ is true, then $T(e, \bar{\sigma})$ is true as well since $\mathcal{E}$ does not yield values in the abort domain. Thus, there must be some $j$ in $\bar{\sigma}$ such that $\bar{\sigma}j = \emptyset$, and therefore, by the assumption on the relationship between $\sigma$ and $\bar{\sigma}$, $\sigma = \emptyset$ and the result for $\mathcal{W}$ is the abort store, $\emptyset$. If $i \overset{2}{=} x$ is false, the result is immediate from the relationship assumed between $\sigma$ and $\bar{\sigma}$.

For the second part of the theorem, assume

$$\mathcal{W}[x := e] \sigma \neq \emptyset$$

By the definition of $\mathcal{W}$, $\sigma \overset{2}{=} \emptyset$ is false; therefore, by assumption, no identifier in $\bar{\sigma}$ has an abort value. Thus, $T(e, \bar{\sigma})$ must be false or $\bot$. If it is $\bot$, then by the definition of $T$, the computation of the value of some identifier in the expression $e$ diverges. However, the expression computation will also diverge under $\mathcal{W}$, and the theorem holds. Otherwise, if $i = x$, then

$$(\mathcal{W}[x := e] \sigma)i = (\delta(\sigma, x, \mathcal{E}[e] \sigma))i$$

$$= \mathcal{E}[e] \sigma$$

by the definition of $\delta$, and

$$\mathcal{V}[x := e] i \bar{\sigma} = (T(e, \bar{\sigma}) \rightarrow \emptyset, \mathcal{E}[e] \bar{\sigma})$$

$$= \mathcal{E}[e] \bar{\sigma}$$
By the assumptions on $\sigma$ and $\bar{\sigma}$,

$$\mathcal{E}[e][\bar{\sigma}] = \mathcal{E}[e][\sigma]$$

If $i \neq x$, the denotation under $\mathcal{V}$ is $\bar{\sigma}i$; the result follows directly from the relationship between $\sigma$ and $\bar{\sigma}$ and the definition of $\delta$.

**Case** $s_1; s_2$ The denotations under $\mathcal{W}$ and $\mathcal{V}$ are as follows:

$$\mathcal{W}[[s_1; s_2]]\sigma = \mathcal{W}[[s_2]](\mathcal{W}[[s_1]]\sigma)$$

$$\mathcal{V}[[s_1; s_2]]i\bar{\sigma} = \mathcal{V}[[s_2]](\lambda j.\mathcal{V}[[s_1]]j\bar{\sigma})$$

By the induction hypothesis, the theorem holds for $s_1$. Thus, the stores used as arguments for $s_2$ satisfy the relationship specified in the theorem. By the induction hypothesis again, the theorem holds for $s_2$ so this case is completed.

The other cases follow similarly. $\square$

### 5.3 Control Dependence

The next stage in the derivation, given in Figure 5.4, introduces the control parameter $\kappa$. The domain specifications for this semantics appears in Figure 5.3. Since $\text{abort}$ statements impact the control dependence relation, the control parameters in this stage are more complex than for structured programs. In the if clause, for example,

<table>
<thead>
<tr>
<th>Semantic Domains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w \in \text{val}^T = \text{val} \cup \top$ (abstract)</td>
</tr>
<tr>
<td>$v \in \text{aval}^T = (\text{val} \cup \emptyset) \cup \top$</td>
</tr>
<tr>
<td>$i, x \in \text{ide}$</td>
</tr>
<tr>
<td>$e, p \in \text{exp}$ (abstract)</td>
</tr>
<tr>
<td>$\kappa \in \text{bool}^T$</td>
</tr>
<tr>
<td>$\sigma \in \text{astore}^T = \text{ide} \rightarrow \text{aval}^T$</td>
</tr>
</tbody>
</table>

**Figure 5.3** Domains for the Control Stage of the Abort Semantics
Semantics:

\[ \begin{align*}
\mathcal{E} : \text{exp} & \rightarrow \text{astore}^T \rightarrow \text{val}^T \\
\mathcal{C} : \text{stmt} & \rightarrow \text{ide} \rightarrow \text{bool}^T \rightarrow \text{astore} \rightarrow \text{aval}^T \\
\mathcal{A}_C : \mathcal{C}[x := e] & = \lambda i\kappa\sigma.\pi \rightarrow x \rightarrow (\kappa \rightarrow (T(e, \sigma) \rightarrow \varnothing, E[e]_\sigma), \perp), \sigma \pi \\
\mathcal{I}_C : \mathcal{C}[\text{if } p \ s_i s_f] & = \lambda i\kappa\sigma.\pi \rightarrow \neg(A(s_i, i) \lor A(s_f, i)) \rightarrow \sigma, i, \\
& \quad \text{let} \\
& \quad \quad \kappa^+ = \neg T(p, \sigma) \land \kappa \land (T(p, \sigma) \rightarrow F, E[p]_\sigma), \\
& \quad \quad \kappa^- = \neg T(p, \sigma) \land \kappa \land (T(p, \sigma) \rightarrow F, \neg E[p]_\sigma), \\
& \quad \quad \kappa^P = T(p, \sigma) \land \kappa \in \\
& \quad \quad \quad (\kappa^P \rightarrow \varnothing, \perp) \sqcup \\
& \quad \quad \quad \neg A(s_i, i) \land \neg AB(s_f, i)) \rightarrow (\kappa^+ \rightarrow \sigma, i, \perp), \mathcal{C}[s_i]i\kappa^+ \sigma \sqcup \\
& \quad \quad \quad \neg A(s_f, i) \land \neg AB(s_f, i)) \rightarrow (\kappa^- \rightarrow \sigma, i, \perp), \mathcal{C}[s_f]i\kappa^- \sigma \\
\mathcal{E}_C : \mathcal{C}[\text{abort}] & = \lambda i\kappa\sigma.\kappa \rightarrow \varnothing, \perp \\
\mathcal{C}_C : \mathcal{C}[s_1; s_2] & = \lambda i\kappa\sigma.\mathcal{C}[s_2]i\kappa(\lambda j.\mathcal{C}[s_1]j\kappa\sigma) \\
\mathcal{W}_C : \mathcal{C}[\text{while } p \ s] & = \lambda i\kappa\sigma.\neg A(s, i) \rightarrow \sigma, i, \\
& \quad \text{let } f = \text{fix}(\lambda w.\lambda i\kappa\sigma. \\
& \quad \quad \text{let} \\
& \quad \quad \quad \kappa^+ = \neg (T(p, \sigma) \lor T(i, \sigma)) \land \kappa \land E[p]_\sigma, \\
& \quad \quad \quad \kappa^- = \neg (T(p, \sigma) \lor T(i, \sigma)) \land \kappa \land \neg E[p]_\sigma, \\
& \quad \quad \quad \kappa^P = (T(p, \sigma) \lor T(i, \sigma)) \land \kappa \in \\
& \quad \quad \quad \quad \text{wik}^+(\lambda j.\mathcal{C}[s]j\kappa^+ \sigma) \sqcup (\kappa^- \rightarrow \sigma, i, \perp) \\
& \quad \quad \quad \quad \sqcup (\kappa^P \rightarrow \varnothing, \perp)) \text{ in} \\
& \quad \quad \text{let} \\
& \quad \quad \quad \kappa^+ = \neg T(p, \sigma) \land \kappa \land E[p]_\sigma, \\
& \quad \quad \quad \kappa^- = \neg T(p, \sigma) \land \kappa \land \neg E[p]_\sigma, \\
& \quad \quad \quad \kappa^P = T(p, \sigma) \land \kappa \in \\
& \quad \quad \quad \text{wik}^+(\lambda j.\mathcal{C}[s]j\kappa^+ \sigma) \sqcup (\kappa^- \rightarrow \sigma, i, \perp) \sqcup \\
& \quad \quad \quad (\kappa^P \rightarrow \varnothing, \perp) \\
\mathcal{F}_C : \mathcal{C}[s; \text{end}] & = \lambda i\kappa\sigma.\mathcal{C}[s]i\kappa\sigma \\
\end{align*} \]

where

\[ T(e, \sigma) = \varnothing \in \{ v \ | \ v = \sigma k, k \in R(e) \} \]

**Figure 5.4** Control Stage for Abort Semantics
there are three values to be merged using the lub function.

\[ \mathcal{C}[\text{if } p \; s_t \; s_f] = \lambda i.\sigma.\neg(\mathcal{A}(s_t, i) \lor \mathcal{A}(s_f, i)) \rightarrow \sigma i, \]

let

\[ \kappa^+ = \neg\mathcal{T}(p, \sigma) \land \kappa \land (\mathcal{T}(p, \sigma) \rightarrow \mathcal{E}[p][\sigma]), \]

\[ \kappa^- = \neg\mathcal{T}(p, \sigma) \land \kappa \land (\mathcal{T}(p, \sigma) \rightarrow \mathcal{F}, \neg\mathcal{E}[p][\sigma]), \]

\[ \kappa^P = \mathcal{T}(p, \sigma) \land \kappa \land \mathcal{E}[p][\sigma] \]

\[ (\kappa^P \rightarrow \circ, \bot) \sqcup \]

\[ ((\neg\mathcal{A}(s_t, i) \land \neg\mathcal{A}(s_f, i)) \rightarrow (\kappa^+ \rightarrow \sigma i, \bot), \mathcal{C}[s_t][i\kappa^+\sigma]) \sqcup \]

\[ ((\neg\mathcal{A}(s_f, i) \land \neg\mathcal{A}(s_f, i)) \rightarrow (\kappa^- \rightarrow \sigma i, \bot), \mathcal{C}[s_f][i\kappa^-\sigma]) \]

The result of the computation and thus the control dependence relation depends not only on the value of the predicate but also on the aborts potentially generated by evaluating the predicate. Thus, there are three control parameters, \( \kappa^P, \kappa^+, \) and \( \kappa^- \). The identifier \( \kappa^P \) denotes true if the control parameter for this if statement, \( \kappa, \) denotes true and if the evaluation of the predicate encounters an abort value. The abort stateus of the predicate is computed separately to correctly generate the abort information in the last step of the derivation. If the computation of the predicate does yield the abort value, the abort value is the value of the demanded identifier. The demand propagation is different in the if clause for abort. In this stage of the standard semantics, the demand flows to the branches of the if statement only if the demanded identifier is assigned on that branch. To accommodate abort statements, however, the demand in the abort semantics is only sent to the branches of the if statement if either the identifier is assigned on that branch or if an abort statement appears on that branch within a statement that assigns a value for the demanded identifier. The predicate \( \mathcal{A}(s_t, i) \), defined in Figure 5.5, is true if an abort statement appears in the statement \( s_t \) as a component of a composition or in an if or while statement containing an assignment for the identifier \( i \).

A similar change to the control parameters is made in the while clause, \( W_C \). The additional complication in the while clause involves aborting the loop if the demanded value from a prior iteration becomes the abort value. This test is not made for the first iteration since the value of the demanded identifier before the loop is only relevant if it is demanded from inside the loop or if the identifier is not assigned within the loop.

In addition, in the if clause, \( I_C \), and the while clause, \( W_C \), the statements in the body are not processed and the predicate is not evaluated unless the demanded
identifier is assigned in the body. Thus, an abort statement in an if statement will only affect computations that request values for identifiers assigned in the body of the if statement. If the evaluation of the predicate expression causes an abort, determined by the predicate \(\mathcal{T}\), the abort value is propagated for the demanded identifier \(i\).

The change in the demand propagation in the if statement described above does not affect the assignment clause, \(A_C\). The demand is sent to a branch if an abort statement is in the branch, even if the identifier is not assigned on that branch. The definition of \(A_B\), however, ensures that either an abort statement or an assignment for the demanded identifier does appear on the branch. Both of these statements stop demand propagation if the control parameter is false. Thus, it is not necessary to stop demand propagation in the assignment clause when the control parameter is false and the identifier in the assignment statement is not the demanded identifier. The following theorem establishes this property by showing that the meaning of any program under the meaning function \(C\) is not the value \(\top\).

**Theorem 5.2** For statement \(s\), identifier \(i\), boolean value \(\kappa\) and store \(\sigma\) such that \(\forall j. \sigma j \neq \top\)

\[C[s]i\kappa\sigma \neq \top\]

**Proof** The proof proceeds by induction on the structure of the statement \(s\).

**Case** \(x := e\) Since the expression evaluation function does not return the value \(\top\), this case follows immediately.
Case if $p \; s_i; \; s_f$  If $i$ is not assigned in the body of the if statement, the result holds based on the assumption about the store $\sigma$. Otherwise, the induction hypothesis gives us that

$$C[s_i]i\kappa\sigma \neq T$$

and

$$C[s_f]i\kappa\sigma \neq T$$

There are several cases to consider. If $\kappa = \text{false}$, then $\kappa^+$ and $\kappa^-$ are both false. The result follows directly from Lemma 5.1. If $\kappa = \text{true}$, at most one of $\kappa^+$ or $\kappa^-$ is true. Thus, the result holds using the induction hypothesis, Lemma 5.1 and the assumption on the store $\sigma$.

Case abort This case follows immediately from the structure of the domain.

Case $s_1; \; s_2$ Using the induction hypothesis, the store $\lambda j.C[s]j\kappa\sigma$ fulfills the assumptions of the theorem. Thus, the induction hypothesis holds for statement $s_2$ and the result is shown.

Case while $p \; s$ This case follows using the same reasoning as in the case of the if statement and fixed-point induction.

\[\square\]

Lemma 5.1  Let $s$ be a statement, $i$ an identifier, and $\sigma$ a store. Then,

1. If $A(s, i)$ is true and $\kappa = \text{false}$ then

$$C[s]i\kappa\sigma = \bot$$

2. If $AB(s, i)$ is true and $A(s, i)$ is false and $\kappa = \text{false}$ then

$$C[s]i\kappa\sigma = \bot$$

3. If $A(s, i)$ is false and $AB(s, i)$ is false, then

$$C[s]i\kappa\sigma = \sigma i$$
Proof The proof proceeds by induction on the structure of the statement \( s \).

Case \( x := e \) If \( A(s, i) \) is be true, \( i = x \) must hold. Part 1 of the theorem follows directly. Otherwise, if \( A(s, i) \) is false, \( i \neq x \) must hold. Part 3 follows directly from the definition of \( C \). Part 2 is vacuously true since \( AB(s, i) \) is false in this case.

Case if \( p \ s \ s_f \) For Part 3, if \( A(s, i) \) is false, then both \( A(s_t, i) \) and \( A(s_f, i) \) are false, by the definition of \( A \). Thus, Part 3 is shown. For Part 1, since \( \kappa \) is false, \( \kappa^+ \) and \( \kappa^- \) are both either false or \( \bot \). The result follows directly if either value is \( \bot \). Otherwise, the result follows using the induction hypothesis and the definition of \( \cup \) and \( C \).

For Part 2, by the definition of \( AB \), for an if statement \( s \), \( AB(i, s) \) can not be true if \( A(s, i) \) is false, so the result is vacuously true.

Case abort Parts 1 and 3 are vacuously true. Part 2 follows directly from the definition of \( C \).

Case \( s_1; s_2 \) If \( A(s, i) \) is false, then both \( A(s_1, i) \) and \( A(s_2, i) \) are false. Part 2 follows using the induction hypothesis for both \( s_1 \) and \( s_2 \). For Part 1, if \( A(s, i) \) is true, \( A(s_1, i) \) is true, \( A(s_2, i) \) is true or both. If \( A(s_2, i) \) is true, Part 1 of the theorem holds using the induction hypothesis for \( s_2 \). Otherwise, \( A(s_1, i) \) must be true and \( A(s_2, i) \) false. By the induction hypothesis, Part 2, the denotation of \( s_2 \) is

\[(\lambda j. C[s_1][j \kappa \sigma])i\]

By the induction hypothesis, Part 1, however, this denotes \( \bot \) since \( A(s_1, i) \) is true.

For Part 2, if \( AB(s, i) \) is true, \( AB(s_1, i) \) is true, \( AB(s_2, i) \) is true, or both are true. The same reasoning holds as in the case of Part 1.

The invariant for the control stage is a modification of that for the demand stage. The control stage further alters the semantics of abort by ignoring abort statements.
associated with demands from a predicate that does not directly impact the value of the demanded identifier. In addition, there is a lifting of the semantics as in the case of the control stage for language $W$.

**Theorem 5.3** For all programs $P$, stores $\sigma$ and $\bar{\sigma}$ such that:

$$\forall j. (\sigma j = \emptyset \supset (\bar{\sigma} j = \emptyset \lor \bar{\sigma} j \subseteq \sigma j)) \land (\bar{\sigma} j \neq \emptyset \supset (\bar{\sigma} j \subseteq \sigma j))$$

for control values $\kappa$, and identifiers $i$,

1) $C[P]i\kappa\sigma = \emptyset \land \kappa \supset \forall \forall P[i\bar{\sigma} = \emptyset \lor \forall \forall P[i\bar{\sigma} = \bot$,

2) $\forall \forall P[i\bar{\sigma} \neq \emptyset \land \kappa \supset \forall \forall P[i\bar{\sigma} \subseteq C[P]i\kappa\sigma$.

**Note:** The condition on the input stores arise from the two ways the meaning of programs differ under the two semantics. In addition to the lifting of the semantics produced by bypassing unnecessary predicate evaluations, the semantics of abort under $C$ may return a value for terms that denote the abort value under $\forall$. The two parts correspond to the two parts in the theorem showing the correspondence between $W$ and $\forall$. Part (1) of the theorem states that, if the abort value results from the $C$ semantics, the $\forall$ semantics either diverges before reaching the abort or also returns the abort value. Part (2) of the theorem applies when the abort value is not the result under the $\forall$ semantics. In this case, as in the case of language $W$, the denotation of the term under the semantics $\forall$ approximates the denotation under $C$. □

**Proof** The proof is by induction on the structure of the program $P$.

**Case** $x := e$ The denotations are

$$\forall[x := e]i\bar{\sigma} = i \overset{2}{=} x \rightarrow (T(e, \bar{\sigma}) \rightarrow \emptyset, \forall[e]i\bar{\sigma}), \bar{\sigma}i$$

and

$$C[x := e]i\kappa\sigma = i \overset{2}{=} x \rightarrow (\kappa \rightarrow (T(e, \sigma) \rightarrow \emptyset, \forall[e]i\sigma), \bot), \sigma i$$

For Part (1) and Part (2), assuming $\kappa$ is true, there are two cases to consider: $x = i$ and $x \neq i$. If $x \neq i$, the result is immediate from the conditions on the stores since

$$\forall[x := e]i\bar{\sigma} = \bar{\sigma}i \text{ and } C[x := e]i\kappa\sigma = \sigma i$$
For Part (1) of the theorem with $x = i$, assume $C[x := e]i\kappa\sigma = \emptyset$. Thus, $T(e, \sigma)$ is true since other values of $T(e, \sigma)$ would violate the assumption. By the conditions on the store, $T(e, \bar{\sigma})$ is either true $\bot$. The theorem holds in either case.

For Part (2) of the theorem, assume $\forall \sigma \neq \emptyset$. $T(e, \bar{\sigma})$ must therefore be false or $\bot$. If it is $\bot$, the theorem follows immediately. Otherwise, by the conditions on the stores, $T(e, \sigma)$ must also be false. Thus, the denotations reduce to the following:

$$\forall[x := e]i\bar{\sigma} = E[e]i\bar{\sigma} \text{ and } C[x := e]i\kappa\sigma = E[e]i\sigma$$

These two expression evaluations are equal given the conditions on the stores.

**Case** $s_1; s_2$ The denotations for statement composition are as follows:

$$\forall[s_1; s_2]i\bar{\sigma} = \forall[s_2]i(\lambda j.\forall[s_1]j\bar{\sigma})$$

and

$$C[s_1; s_2]i\kappa\sigma = C[s_2]i\kappa(\lambda j.C[s_1]j\kappa\sigma)$$

If $\kappa$ is false, the theorem is vacuously true, so assume that $\kappa$ is true. To invoke the induction hypothesis for statement $s_2$, we need to show that the stores $\sigma' = \lambda j.C[s_1]j\kappa\sigma$ and $\bar{\sigma}' = \lambda j.\forall[s_1]j\bar{\sigma}$ satisfy the conditions stated in the theorem.

Since $\sigma$ and $\bar{\sigma}$ satisfy the conditions of the theorem and $\kappa$ is true, for identifier $j$,

$$C[s_1]j\kappa\sigma = \emptyset \supset \forall[s_1]i\sigma = \emptyset \lor \forall[s_1]i\bar{\sigma} \subseteq C[s_1]i\kappa\sigma$$

and

$$\forall[s_1]i\bar{\sigma} \neq \emptyset \supset \forall[s_1]i\bar{\sigma} \subseteq C[s_1]i\kappa\sigma$$

hold by the induction hypothesis. Thus, $\sigma'$ and $\bar{\sigma}'$ satisfy the conditions of the theorem if $\kappa$ is true. Therefore, by the induction hypothesis, the theorem holds for $s_2$ in the updated stores and thus holds for $s_1; s_2$. 
Case if \( p \ s_t \ s_f \) The denotations for the if statements are as follows:

\[
\forall [\text{if } p \ s_t \ s_f]i\bar{\sigma} = T(p, \bar{\sigma}) \rightarrow \emptyset, (E[p]\bar{\sigma} \rightarrow \forall [s_t]i\bar{\sigma}, \forall [s_f]i\bar{\sigma})
\]

and

\[
C[\text{if } p \ s_t \ s_f] = \lambda \kappa \sigma.\neg(t(A(s_t, i) \lor A(s_f, i)) \rightarrow \sigma i, \text{ let } \\
\kappa^+ = \neg T(p, \sigma) \land \kappa \land E[p]\sigma, \\
\kappa^- = \neg T(p, \sigma) \land \kappa \land \neg E[p]\sigma, \\
\kappa^P = T(p, \sigma) \land \kappa \text{ in } \\
(\kappa^P \rightarrow \emptyset, \bot) \cup \\
((\neg A(s_t, i) \land \neg A_{\text{run}}(s_t, i)) \rightarrow \\
(\kappa^+ \rightarrow \sigma i, \bot), C[s_t]i\kappa^+\sigma) \\
\cup \\
((\neg A(s_f, i) \land \neg A_{\text{run}}(s_f, i)) \rightarrow \\
(\kappa^- \rightarrow \sigma i, \bot), C[s_f]i\kappa^-\sigma)
\]

There are two cases to consider: whether \( A(s_t, i) \lor A(s_f, i) \) is true or false. If it is false, the demanded identifier is not assigned in the body of the if statement. The result for \( C \) is simply \( \sigma i \). The situation is more complex for \( V \) however. In addition to the potential for divergence in the predicate \( p \), an abort can occur. Thus, the result for \( V \) is either \( \bot \) for divergence in the predicate, the abort value, or \( \bar{\sigma} i \).

For Part (1) of the theorem, assume \( C[P]i\kappa\sigma = \emptyset \). Since \( C[P]i\kappa\sigma = \sigma i \), the conditions on the stores require that \( \bar{\sigma} i \not\in T \), so the theorem holds if the denotation for \( V[P]i\sigma \) is \( \bar{\sigma} i \) or if an abort occurs. If the predicate expression diverges, the theorem holds since \( \bot \) approximates all other values in the domain.

For Part (2) of the theorem, assume \( V[P]i\sigma \neq \emptyset \). The theorem follows directly from the conditions on the input store.

In the other case, \( A(s_t, i) \lor A(s_f, i) \) is true. For Part (1) of the theorem, we again assume

\[
C[P]i\kappa\sigma = \emptyset
\]
Thus, either $T(p, \sigma)$ is true or the branch enabled by the predicate expression denotes the abort value or $\sigma i = \emptyset$. Since we assume for this part of the theorem that the result is $\emptyset$, $T(p, \sigma)$ can not be $\bot$. By the conditions on the stores, if $T(p, \sigma)$ is true, $T(p, \bar{\sigma})$ is also true or $\bot$ and the theorem holds in this case.

For $T(p, \sigma)$ false, assume without loss of generality that $\kappa^+$ is true. Thus, $E[p] \sigma$ is true. By the conditions on the store, $E[p] \bar{\sigma}$ is true, or the expression evaluation diverges, or $T(p, \bar{\sigma})$ is true. The theorem follows directly if the expression evaluation diverges or if $T(p, \bar{\sigma})$ is true. Otherwise, if $E[p] \bar{\sigma}$ is true, then

$$V[\text{if } p \ s_t \ s_f \| \bar{\sigma}] = V[s_t] \bar{\sigma}$$

Since $\kappa^+$ is true, $\kappa^-$ and $\kappa^P$ are false. The first component on the least upper bound operation is clearly $\bot$. The third component is $\bot$ by Lemma 5.1. Therefore, the denotation for $C$ becomes

$$(-A(s_t, i) \land \neg AB(s_t, i)) \rightarrow (\kappa^+ \rightarrow \sigma i, \bot), C[s_t]i\kappa^+\sigma$$

If $\neg A(s_t, i) \land \neg AB(s_t, i)$ is true, the denotation becomes $\sigma i$, and $i$ is not assigned in $s_t$. By assumption, $\sigma i$ is the abort value. If no aborts occur in $V[s_t]i\bar{\sigma}$, the final value is found in the store $\bar{\sigma}$. The theorem then follows from the conditions on the stores. If an abort occurs or if the evaluation diverges, the theorem also holds. Since $i$ is not assigned in $s_t$, these are the only possible values for $V$, using the same reasoning as in the standard language proof. Thus, the theorem holds for this sub-case.

If $\neg A(s_t, i) \land \neg AB(s_t, i)$ is false, the theorem holds by the induction hypothesis.

For Part (2) of the theorem, we assume

$$V[P]i\bar{\sigma} \neq \emptyset$$

For this case, we know that $T(p, \bar{\sigma})$ is not true. If it is false, by the assumptions on the stores, $T(p, \sigma)$ must also be false. If it is $\bot$, the result follows immediately. The rest of the reasoning for this case is the same as for the standard language and is omitted.

The rest of the cases follow similarly.
5.4 Program Dependence Trees in the Presence of Abort

The next stage of the derivation abstracts over the initial store to prepare for the split of the function into the static and dynamic components. Figures 5.6 and 5.7 present the relevant clauses. This transformation proceeds as in the previous derivations. For example, the assignment clause, \( A_{C'} \), incorporates the partial store parameter \( \gamma \). The function \( T' \) provides the same information as the \( T \) function in the previous stages. The other clauses include analogous modifications.

As in the previous languages, the semantics is preserved by this step in the derivation.

**Theorem 5.4** For all programs \( P \), stores \( \sigma \), \( \bar{\sigma} \) and \( \gamma \) such that

\[
\forall j. \bar{\sigma} j = \gamma j \sigma
\]

control parameters \( \kappa \) and \( \bar{\kappa} \) such that

\[
\bar{\kappa} = \kappa \sigma
\]

Semantic Domains:

\[
\begin{align*}
w & \in val^T = val \cup T \quad \text{(abstract)} \\
v & \in aval^T = (val \cup \emptyset) \cup T \\
i, x & \in ide \\
e, p & \in exp \quad \text{(abstract)} \\
\kappa & \in (\text{store} \to \text{bool}^T) \\
\gamma & \in (\text{ide} \to \text{store} \to aval^T) \\
\sigma & \in \text{store} = \text{ide} \to val
\end{align*}
\]

Semantics:

\[
\begin{align*}
\mathcal{E} & : \exp \to astore^T \to val^T \quad \text{(abstract)} \\
\mathcal{C'} & : \text{stmt} \to \text{ide} \to (\text{store} \to \text{bool}^T) \to (\text{ide} \to \text{store} \to aval^T) \to \text{store} \to aval^T
\end{align*}
\]

**Figure 5.6** Domains for the Abstraction Stage of Abort Semantics
\[
\begin{align*}
A_{C'}: \quad C'[x := e] &= \lambda i k \gamma. i \geq 2 \ x \rightarrow (\lambda \sigma. k \sigma \rightarrow (T'(e, \gamma, \sigma) \rightarrow \emptyset, E[e](\lambda j. \gamma j \sigma)), \bot), \\
&\quad \quad \lambda \sigma. \gamma \sigma \\
I_{C'}: \quad C'[\text{if } p \ s_1 \ s_2] &= \lambda i k \gamma. \neg(A(s_1, i) \lor A(s_2, i)) \rightarrow (\lambda \sigma. k \sigma \rightarrow \gamma i \sigma, \bot), \\
&\quad \quad \text{let} \\
&\quad \quad \quad \kappa^+ = \lambda \sigma. \neg T'(p, \gamma, \sigma) \land k \sigma \land E[p](\lambda j. \gamma j \sigma), \\
&\quad \quad \quad \kappa^- = \lambda \sigma. \neg T'(p, \gamma, \sigma) \land k \sigma \land E[p](\lambda j. \gamma j \sigma), \\
&\quad \quad \quad \kappa^P = \lambda \sigma. T'(p, \gamma, \sigma) \land k \sigma \\
&\quad \quad \quad \lambda \sigma. (\kappa^P \sigma \rightarrow \emptyset, \bot) \\
&\quad \quad \quad \quad \cup \\
&\quad \quad \quad \quad ((\neg A(s_1, i) \land \neg AB(s_1, i)) \rightarrow \\
&\quad \quad \quad \quad (\lambda \sigma. k^+ \sigma \rightarrow \gamma i \sigma, \bot), C'[s_1]i k^+ \gamma) \\
&\quad \quad \quad \quad \cup \\
&\quad \quad \quad ((\neg A(s_2, i) \land \neg AB(s_2, i)) \rightarrow \\
&\quad \quad \quad \quad (\lambda \sigma. k^- \sigma \rightarrow \gamma i \sigma, \bot), C'[s_2]i k^- \gamma) \\
E_{C'}: \quad C'[\text{abort}] &= \lambda i k \gamma.l \sigma. k \sigma \rightarrow \emptyset, \bot \\
C_{C'}: \quad C'[s_1; s_2] &= \lambda i k \gamma. C'[s_2]i k(\lambda j. C'[s_1]j k \gamma \\
W_{C'}: \quad C'[\text{while } p \ s] &= \lambda i k \gamma. i \neg A(s, i) \rightarrow (\lambda \sigma. \gamma i \sigma), \\
&\quad \quad \text{let } f = \text{fix}(\lambda w. l i k \sigma, \\
\quad \quad \quad \text{let} \\
\quad \quad \quad \quad \kappa^+ = \lambda \sigma. \neg (T'(p, \gamma, \sigma) \lor T'(i, \gamma, \sigma)) \\
\quad \quad \quad \quad \land k \sigma \land E[p](\lambda j. \gamma j \sigma), \\
\quad \quad \quad \quad \kappa^- = \lambda \sigma. \neg (T'(p, \gamma, \sigma) \lor T'(i, \gamma, \sigma)) \\
\quad \quad \quad \quad \land k \sigma \land E[p](\lambda j. \gamma j \sigma), \\
\quad \quad \quad \quad \kappa^P = \lambda \sigma. (T'(p, \gamma, \sigma) \lor T'(i, \gamma, \sigma)) \\
\quad \quad \quad \quad \land k \sigma \\
\quad \quad \quad \quad \text{in} \\
\quad \quad \quad \quad (\lambda j. C'[s]j k^+ \gamma) \cup (\lambda \sigma. k^- \sigma \rightarrow \gamma i \sigma, \bot) \\
\quad \quad \quad \quad \cup (\lambda \sigma. k^P \sigma \rightarrow \emptyset, \bot) \\
F_{C'}: \quad C'[s; \text{end}] &= \lambda i k \gamma. C[s]i k \gamma \\
\text{where} \\
T'(e, \gamma, \sigma) &= \emptyset \in \{ v \mid v = \gamma j \sigma, j \in R(e) \}
\end{align*}
\]

Figure 5.7 Meaning Functions for the Abstraction Stage of Abort Semantics
and all identifiers $i$,

$$C[P]iT\bar{\sigma} = C'[P](\lambda \sigma. T)(\lambda i\sigma. i\sigma)\sigma$$

**Proof** The proof proceeds by induction on the structure of the term $P$.

The only complication introduced by the **abort** statement is the predicates $T$ and $T'$. For expression $e$, we must show that

$$T(e, \bar{\sigma}) = T'(e, \gamma, \sigma)$$

By the definitions of $T$ and $T'$,

$$T(e, \bar{\sigma}) = \varnothing \in \{v \mid v = \bar{\sigma}k, k \in \mathcal{R}(e)\}$$

and

$$T'(e, \gamma, \sigma) = \varnothing \in \{v \mid v = \gamma k\sigma, k \in \mathcal{R}(e)\}$$

By the assumption on $\gamma$, $\sigma$, and $\bar{\sigma}$, for all identifiers $k \in \mathcal{R}(e)$, $\gamma k\sigma = \bar{\sigma}k$

so the two predicates are equal. The rest of the proof follows the proof strategy for the core language and is omitted. \hfill \square

The final stage of the derivation splits the $C'$ function into a static portion, $\mathcal{G}$, and a dynamic portion, $\mathcal{P}$. Figure 5.9 and Figure 5.10 respectively give these functions; the semantic domains for these functions appear in Figure 5.8. Several changes from the previous $\mathcal{G}$ functions are required to support the **abort** construct. The **abort** clause, $E_\mathcal{G}$, creates an **abort** node for the **abort** statement. This node serves as the signal for the creation of the **abort** related control dependences.

In the **if** clause there are now three types of control nodes to support the three different values possible from an **if** statement:

$$\mathcal{G}[\text{if } p \ s_t \ s_f] = \lambda i\kappa \gamma. (A(s_t, i) \lor A(s_f, i)) \rightarrow \gamma i,$$

let

$$\kappa^+ = \langle \mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), p, \{(j, \gamma j) \mid j \in \mathcal{R}(p)\} \rangle,$$

$$\kappa^- = \langle \mathcal{F}, \kappa, \mathcal{Q}(p, \gamma), p, \{(j, \gamma j) \mid j \in \mathcal{R}(p)\} \rangle,$$

$$\kappa' = \langle \mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), p \rangle \text{ in } \langle \kappa^P, \rangle,$$

$$\neg A(s_t, i) \land \neg AB(s_t, i) \rightarrow$$

$$\langle \kappa^+, \mathcal{Q}(i, \gamma), i, \{(i, \gamma i)\} \rangle, \mathcal{G}[s_t]i\kappa^+\gamma,$$

$$\neg A(s_f, i) \land \neg AB(s_f, i) \rightarrow$$

$$\langle \kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma i)\} \rangle, \mathcal{G}[s_f]i\kappa^-\gamma.$$
Semantic Domains:

\[
\begin{align*}
  w & \in \text{val} & = \text{val} \\
  v & \in \text{aval}^T & = (\text{val} \oplus \ominus) \cup \top \\
  i, x & \in \text{ide} \\
  e, p & \in \text{exp} \\
  \sigma & \in \text{store} & = \text{ide} \rightarrow \text{val} \\
  c_e, c_p & \in \text{code-tbl} & = \text{ide} \leftrightarrow \text{code} \\
  c & \in \text{code} & = \text{d-node} \oplus (\text{d-node} \odot \text{d-node} \odot \text{d-node}) \\
  & & \oplus \text{d-node}^+ \\
  c_d & \in \text{d-node} & = (\text{p-node} \odot \text{p-node}^* \odot \text{exp} \odot \text{code-tbl}) \oplus \\
  & & (\text{p-node} \odot \text{ide}) \oplus (\text{ide} \odot \text{code}) \oplus (\text{p-node}) \\
  \kappa, c_\kappa & \in \text{p-node} & = \text{true}_c \oplus \text{false}_c \oplus \\
  & & (\text{p-node} \odot \text{p-node}^* \odot \text{exp} \odot \text{code-tbl}) \oplus \\
  & & (\text{p-node} \odot \text{p-node}^* \odot \text{exp} \odot \text{code-tbl}) \oplus \\
  & & (\text{p-node} \odot \text{p-node}^*)
\end{align*}
\]

Semantics:

\[
\begin{align*}
  \mathcal{G} & : \text{update} \rightarrow \text{stmt} \rightarrow \text{ide} \rightarrow \text{p-node} \rightarrow \text{code-tbl} \rightarrow \text{code} \\
  \mathcal{E} & : \text{exp} \rightarrow \text{store}^T \rightarrow \text{val}^T \quad \text{(abstract)} \\
  \mathcal{P} & : \text{code} \rightarrow \text{store}^T \rightarrow \text{aval}^T \\
  \mathcal{P}_k & : \text{p-node} \rightarrow \text{store}^T \rightarrow \text{bool}^T \\
  \mathcal{P}_a & : \text{p-node}^* \rightarrow \text{store}^T \rightarrow \text{bool}^T
\end{align*}
\]

Figure 5.8 Domains for PDG Functions
\[ A_G : \mathcal{G}[x := e] = \lambda i. \gamma_i \mapsto x \rightarrow (\kappa, \mathcal{Q}(e, \gamma), e, \mathcal{M}(e, \gamma), \gamma_i) \]

\[ I_G : \mathcal{G}[\text{if } p \text{ st } s_f] = \lambda i. \gamma_i \rightarrow (A(s_t, i) \lor A(s_f, i)) \rightarrow \gamma_i, \]

let

\[ \kappa^+ = (T, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^- = (F, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^P = (I, \kappa, \mathcal{Q}(p, \gamma)) \text{ in} \]
\[ \langle \kappa^P \rangle, \]
\[ ((\neg A(s_t, i) \land \neg AB(s_t, i)) \rightarrow \]
\[ \langle \kappa^+, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\} \rangle, \mathcal{G}[s_i] i k^+ \gamma, \]
\[ ((\neg A(s_f, i) \land \neg AB(s_f, i)) \rightarrow \]
\[ \langle \kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\} \rangle, \mathcal{G}[s_f] i k^- \gamma) \]

\[ A_G : \mathcal{G}[\text{abort}] = \lambda i. \gamma_i.(k) \]

\[ C_G : \mathcal{G}[s_1; s_2] = \lambda i. \gamma_i.\mathcal{G}[s_2] i k (\lambda j. \mathcal{G}[s_1] i k^+ \gamma) \]

\[ W_G : \mathcal{G}[\text{while } p \text{ s}] = \lambda i. \gamma_i.\neg A(s, i) \rightarrow \gamma_i, \]

let \( f = \text{fix}(\lambda w. \lambda i. \gamma_i) \)

let

\[ \kappa^+ = (T, \kappa, [\mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma)], p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^- = (F, \kappa, [\mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma)], p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^P = (I, \kappa, [\mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma)], p) \text{ in} \]
\[ \langle \kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\} \rangle \circ \langle \kappa^P \rangle \circ \]
\[ \text{wix}^+(\lambda j. \mathcal{G}[s] j k^+ \gamma) \text{ in} \]

\[ \kappa^+ = (T, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^- = (F, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}(p, \gamma)) \]
\[ \kappa^P = (I, \kappa, \mathcal{Q}(p, \gamma), p) \text{ in} \]
\[ \langle \kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\} \rangle \circ \langle \kappa^P \rangle \circ \]
\[ \text{fix}^+(\lambda j. \mathcal{G}[s] j k^+ \gamma) \]

\[ F_G : \mathcal{G}[s; \text{end}] = \lambda i. \gamma_i.(i, \mathcal{G}[s] i k^+ \gamma) \]

where

\[ \mathcal{Q}(p, \gamma) = [k_1, \ldots] \mid k_i \in \{F(k) \mid \langle k \rangle \in \mathcal{R}(p), k \in \text{p-node}\} \]

\[ \mathcal{M}(e, \gamma) = \{(j, \gamma_j) \mid j \in \mathcal{R}(e)\} \]

\[ F(\text{true}_e) = \text{false}_e \]
\[ F((T, c_k, c_a, p, c_p)) = \langle F, c_k, c_a, p, c_p \rangle \]
\[ F((F, c_k, c_a, p, c_p)) = \langle T, c_k, c_a, p, c_p \rangle \]

**Figure 5.9** \( \mathcal{G} \) Function for the Abort Semantics
The if node in the pdt contains three components; the component for the true and false branches as in the earlier $\mathcal{G}$ functions, and the additional component with the abort information for the predicate. The function $\mathcal{Q}(p, \gamma)$, extracts from $\gamma$ the abort nodes that may return an abort value for the identifiers in expression $p$. This set represents the additional control dependences present for abort statements in the program. The function $\mathcal{F}$ switches the truth value of the control node representing the abort conditions. This switch is needed since the abort occurs if the control node found in $\gamma$ is true, and the node should execute normally if the control node is false. Thus, the tag for the control dependence is the opposite of the control node in the abort node.

The assignment clause includes a component containing the control dependences for abort statements relevant to its expression $e$:

$$\mathcal{G}[x := e] = \lambda i, \gamma. i \vdash x \rightarrow (\kappa, \mathcal{Q}(e, \gamma), e, \{(j, \gamma_j) \mid j \in \mathcal{R}(e)\}, \gamma_i$$

Finally, the while clause creates the infinite series of branches representing the different possible evaluations of the loop:

$$\mathcal{G}[\text{while } p \mathcal{Q} s] = \lambda i, \gamma. \neg A(s, i) \rightarrow \gamma_i,$n
let $f = \text{fix}((\lambda w. \lambda i, \gamma. \mathcal{G}[w \mathcal{Q} s]) \circ (\kappa^P) \circ \text{wikt}((\lambda j. \mathcal{G}[s] j\kappa^+ \gamma))$ in

let

$$\kappa^+ = (\mathcal{T}, \kappa, [\mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma)], p, \{(j, \gamma_j) \mid j \in \mathcal{R}(p)\}),$$

$$\kappa^- = (\mathcal{F}, \kappa, [\mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma)], p, \{(j, \gamma_j) \mid j \in \mathcal{R}(p)\}),$$

$$\kappa^P = (\mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), \mathcal{Q}(i, \gamma), p) \text{ in}$$

$$(\kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\}) \circ (\kappa^P) \circ \text{wikt}((\lambda j. \mathcal{G}[s] j\kappa^+ \gamma))$$

let

$$\kappa^+ = (\mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), p, \{(j, \gamma_j) \mid j \in \mathcal{R}(p)\}),$$

$$\kappa^- = (\mathcal{F}, \kappa, \mathcal{Q}(p, \gamma), p, \{(j, \gamma_j) \mid j \in \mathcal{R}(p)\}),$$

$$\kappa^P = (\mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), p) \text{ in}$$

$$(\kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma_i)\}) \circ (\kappa^P) \circ \text{wikt}((\lambda j. \mathcal{G}[s] j\kappa^+ \gamma))$$

As in the if clause, each component has three parts, for the true, false, and abort information. As in the control semantics, the abort information in the first iteration of the while sequence contains only abort nodes for the predicate expression. The abort information for the rest of the iterations contains abort nodes from the demand propagation in the prior iterations in addition to the abort information for the predicate.
The function $\mathcal{P}$, shown in Figure 5.10, includes many changes to process the valve nodes, the abort node, and the abort control information.

The clause for the assignment node implements the assignment clause from $\mathcal{C}'$:

$$\mathcal{P}[(c_k, c_a, e, c_e)] = \lambda \sigma. \mathcal{P}_k[c_k] \sigma \rightarrow (\mathcal{P}_a[c_a] \sigma \rightarrow E[e] RS(c_e, \sigma), \varnothing), \bot$$

If the control node is found to be true by the $\mathcal{P}_k$ function, the abort information for the expression is checked by the $\mathcal{P}_a$ function. If the abort information is false, then an abort occurred in the evaluation of the identifiers for the expression and the abort value is the value of the node. The checks for the abort information is reversed from the function $\mathcal{C}'$ to reflect the usual control dependence information for pdgs. Otherwise, the expression $e$ is evaluated and this value is the result. If the control node is false, $\bot$ is returned, as in the $\mathcal{P}$ function for the standard semantics.

The abort node clause returns the abort value if the control node is true; otherwise, $\bot$ is the result of the node:

$$\mathcal{P}[(c_k)] = \lambda \sigma. \mathcal{P}_k[c_k] \sigma \rightarrow \varnothing, \bot$$

---

\[
\begin{align*}
\mathcal{P}[(c_k, c_a, e, c_e)] &= \lambda \sigma. \mathcal{P}_k[c_k] \sigma \rightarrow (\mathcal{P}_a[c_a] \sigma \rightarrow E[e] RS(c_e, \sigma), \varnothing), \bot \\
\mathcal{P}[(c_k, t)] &= \lambda \sigma. \mathcal{P}_k[c_k] \sigma \rightarrow \sigma i, \bot \\
\mathcal{P}[(t, c)] &= \lambda \sigma. \mathcal{P}_k[c_t] \sigma \\
\mathcal{P}[(c_k)] &= \lambda \sigma. \mathcal{P}_k[c_k] \sigma \rightarrow \varnothing, \bot \\
\mathcal{P}[(c_a, c_t, c_f)] &= \lambda \sigma. \mathcal{P}_k[c_a] \sigma \sqcup \mathcal{P}[c_t] \sigma \sqcup \mathcal{P}[c_f] \sigma \\
\mathcal{P}[(c_t, c_1, \ldots)] &= \lambda \sigma. \mathcal{P}_k[c_t] \sigma \sqcup \mathcal{P}[c_1, \ldots] \sigma
\end{align*}
\]

\[
\begin{align*}
\mathcal{P}_k[(\text{true}_c)] &= \lambda \sigma. T \\
\mathcal{P}_k[(\text{false}_c)] &= \lambda \sigma. F \\
\mathcal{P}_k[(T, c_k, c_a, p, c_p)] &= \lambda \sigma. \mathcal{P}_a[c_a] \sigma \rightarrow \mathcal{P}_k[c_k] \sigma \land E[p] RS(c_p, \sigma), F \\
\mathcal{P}_k[(F, c_k, c_a, p, c_p)] &= \lambda \sigma. \mathcal{P}_a[c_a] \sigma \rightarrow \mathcal{P}_k[c_k] \sigma \land \neg E[p] RS(c_p, \sigma), F \\
\mathcal{P}_k[(\Gamma, c_k, c_a)] &= \lambda \sigma. \mathcal{P}_k[c_k] \sigma \land \neg \mathcal{P}_a[c_a] \sigma
\end{align*}
\]

\[
\begin{align*}
\mathcal{P}_a[\emptyset] &= \lambda \sigma. T \\
\mathcal{P}_a[(c_k)] &= \lambda \sigma. \mathcal{P}_k[c_k] \sigma \\
\mathcal{P}_a[(c_1, c_2, \ldots)] &= \lambda \sigma. \mathcal{P}_k[c_1] \sigma \land \mathcal{P}_a[(c_2, \ldots)] \sigma
\end{align*}
\]

$$RS(C_e, \sigma) = \{(b, \mathcal{P}[C_b] \sigma) \mid (b, C_b) \in C_e\}$$

---

**Figure 5.10** Final Stage for Abort Semantics
The if node clause evaluates the three components and takes the lub of the three results:

\[ \mathcal{P}[[c_2, c_t, c_f]] = \lambda \sigma. \mathcal{P}[[c_a]] \sigma \sqcup \mathcal{P}[[c_t]] \sigma \sqcup \mathcal{P}[[c_f]] \sigma \]

The components for the true and the false branch include in their control node the abort information of the predicate. The abort component only includes this abort information. The while node clause is unchanged.

The function \( \mathcal{P}_k \) includes changes for the abort information as well. The clause to process the T control node, for example,

\[ \mathcal{P}_k[[\langle T, c_k, c_a, p, c_p \rangle]] = \lambda \sigma. \mathcal{P}_a[[c_a]] \sigma \rightarrow \mathcal{P}_k[[c_k]] \sigma \land \mathcal{E}[p] \mathcal{R}c_p(c_p, \sigma), F \]

first checks the abort component of the node using the function \( \mathcal{P}_a \). If the abort component is false, an enabled abort node was processed. Thus, the control node returns false. Otherwise, the check of the predicate expression is performed as in the standard semantics. The analogous change is made in the F clause.

A clause is added to the \( \mathcal{P}_k \) function to process the abort component of the if clause. This clause,

\[ \mathcal{P}_k[[\langle I, c_k, c_a, p \rangle]] = \lambda \sigma. \mathcal{P}_k[[c_k]] \sigma \land \neg \mathcal{P}_a[[c_a]] \sigma \]

returns true if the abort information is false and the control parameter is true. This control node appears as the abort component of the if node. Thus, the abort value is returned from an if node if the node itself is enabled by its control information, \( c_k \), and if the abort information for the predicate, \( c_a \), is false.

A new function, \( \mathcal{P}_a \), processes the sets of abort control dependence predecessors. Since a node may have no abort control dependence predecessors, the clause for the empty set always returns true to indicate not to abort the evaluation. If several predecessors exist, the logical and of the individual values is used. Thus, if an abort occurs for any of the demands, the abort information is true. The \( \mathcal{P}_k \) function interprets the individual control nodes in the set.

As in the earlier languages, the initial code tree argument creates the iddef nodes for any identifiers that are used before being assigned in the program. With this initial parameter, the functions \( \mathcal{G}, \mathcal{P} \) and \( C' \) satisfy the invariant specified in the following theorem.

**Theorem 5.5** For all programs \( P \), identifiers \( i \), and stores \( \sigma \),

\[ \mathcal{P}(\mathcal{G}[P][i; \text{true}_{c\gamma_0}] \sigma) = C'[P][i(\lambda \sigma. T)(\lambda j \sigma. \sigma j)] \sigma \]
Proof To prove the correspondence, we prove a stronger lemma, Using Lemma 5.2, the theorem follows since the initial parameters satisfy the requirements of the lemma. The proof of this lemma appears below.

Lemma 5.2 Let $P$ be a program, $\sigma$ a store, $\gamma$ and $\bar{\gamma}$ code trees, $i$ an identifier, and $\kappa$ and $\bar{\kappa}$ control parameters. If

1) $\forall j. \bar{\gamma}j\sigma = P[\bar{\gamma}j]\sigma$
2) $\bar{\kappa}\sigma = P_{\kappa}[\bar{\kappa}]\sigma$

then

$P(G[P]i\kappa\gamma)\sigma = C'[P]i\bar{\kappa}\bar{\gamma}\sigma$

Proof The proof proceeds by induction on the structure of $P$.

Case $x := e$ There are two cases to consider here. First, if $i \neq x$,

$P(G[P]i\kappa\gamma)\sigma = P[\gamma i]\sigma$

and

$C'[P]i\bar{\kappa}\bar{\gamma}\sigma = \gamma i\sigma$

By the assumptions on the inputs, these two terms are equal.

If $i = x$, the applicable denotations are as follows:

$P(G[P]i\kappa\gamma)\sigma = P[(\kappa, Q(e, \gamma), e, MS(e, \gamma))]\sigma$

$= P_{\kappa}[\kappa]\sigma \rightarrow (P_{\kappa}[Q(e, \gamma)]\sigma \rightarrow$

$E[e]\{(j, P[\gamma j]\sigma) \mid j \in R(e)\}$,

$\varnothing), \bot$

and

$C'[P]i\bar{\kappa}\bar{\gamma}\sigma = \bar{\kappa}\sigma \rightarrow (T'(e, \bar{\gamma}, \sigma) \rightarrow \varnothing$,

$E[e](\lambda j. \bar{\gamma}j\sigma)), \bot$

There are several cases to consider here. First, if $\bar{\kappa}\sigma$ diverges, by the assumption on the inputs, $P_{\kappa}[\kappa]\sigma$ also diverges so the resulting terms are equal. If $\bar{\kappa}\sigma$ is false, so is $P_{\kappa}[\kappa]\sigma$ by the input assumptions, so the resulting denotations are both $\varnothing$. If $\bar{\kappa}\sigma$ is true, $P_{\kappa}[\kappa]\sigma$ is also
true. There are two cases to consider. Assume first that $T'(e, \gamma, \sigma)$ is true. Therefore,

$$C'[P][i\bar{k}\gamma] = \emptyset$$

By Lemma 5.3 shown below, $\neg \mathcal{P}_a[Q(e, \gamma)]\sigma$, but if

$$\neg \mathcal{P}_a[Q(e, \gamma)]\sigma$$

holds, then

$$\mathcal{P}[G[P][i\kappa\gamma]]\sigma = \emptyset$$

and the theorem holds.

If $T'(e, \gamma, \sigma)$ is $\perp$, by Lemma 5.3, $\mathcal{P}_a[Q(e, \gamma)]\sigma$ is also $\perp$ and the theorem holds here as well.

Now, assume $T'(e, \gamma, \sigma)$ is false. Therefore,

$$C'[P][i\bar{k}\gamma] = \mathcal{E}[e](\lambda j.\gamma j)\sigma$$

Once again by Lemma 5.3, $\mathcal{P}_a[Q(e, \gamma)]\sigma$ holds, so

$$\mathcal{P}(G[P][i\kappa\gamma])\sigma = \mathcal{E}[e][\{(j, P[j\gamma] j) \mid j \in R(e)\}]$$

By the assumptions on the stores and the fact that the expression evaluation for $e$ depends only on store values for identifiers in $R(e)$, these terms are equal. Thus, the theorem holds for assignment statements.

**Case $s_1; s_2$** By the assumptions on the inputs and the induction hypothesis,

$$\forall j. C'[s_1][j\bar{k}\gamma] = \mathcal{P}(G[s_1][j\kappa\gamma])\sigma$$

Therefore, the inputs to the denotations

$$C'[s_1; s_2][i\bar{k}\gamma] = C'[s_2][i\bar{k}(\lambda j. C'[s_1][j\bar{k}\gamma])]$$

and

$$\mathcal{P}(G[s_1; s_2][i\kappa\gamma]) = \mathcal{P}(G[s_2][i\kappa(\lambda j. G[s_1][j\kappa\gamma])])\sigma$$

satisfy the assumptions of the theorem. Thus, the induction hypothesis applies for statement $s_2$ and the theorem holds for statement composition.
Case if $p \, s_t, s_f$ As with the assignment clause, there are two cases to consider. First, if the identifier $i$ is not assigned in the body of the if statement, the denotations are

$$C'[P]i\tilde{\gamma}\sigma = \tilde{i}\sigma$$

and

$$\mathcal{P}(G[P]i\kappa\gamma)\sigma = \mathcal{P}[\gamma i]\sigma$$

These terms are equal by the assumptions on the input stores. If the identifier is assigned within the body of the if, there are several cases to consider. First, consider the control parameters in the two denotations, using $\kappa^+$ and $\tilde{\kappa}^+$ as examples:

$$\tilde{\kappa}^+\sigma = \neg T'(p, \bar{\gamma}, \sigma) \land \tilde{\kappa}\sigma \land E[p](\lambda j.\bar{\gamma}j\sigma)$$

and

$$\mathcal{P}_k[\kappa^+]\sigma = \mathcal{P}_k[(\mathcal{T}, \kappa, \mathcal{Q}(p, \gamma), p, \{(j, \gamma j) \mid j \in \mathcal{R}(p)\})]\sigma$$

$$= \mathcal{P}_a[\mathcal{Q}(p, \gamma)]\sigma \rightarrow$$

$$\mathcal{P}_k[K]\sigma \land E[p]\{(j, \mathcal{P}[\gamma j]) \mid j \in \mathcal{R}(p)\}, \mathcal{F}$$

By the assumptions on the input stores and the properties of expression evaluation,

$$E[p](\lambda j.\bar{\gamma}j\sigma) = E[p]\{(j, \mathcal{P}[\gamma j]) \mid j \in \mathcal{R}(p)\}$$

and by the assumptions on the input control parameters,

$$\tilde{\kappa}\sigma = \mathcal{P}_k[\kappa]\sigma$$

Finally, if $T'(p, \bar{\gamma}, \sigma)$ is true, $\neg \mathcal{P}_a[\mathcal{Q}(p, \gamma)]\sigma$ must hold by Lemma 5.3 and both control parameters are false. By Lemma 5.3, if one expression diverges, so does the other. If $T'(p, \bar{\gamma}, \sigma)$ is false, $\mathcal{P}_a[\mathcal{Q}(p, \gamma)]\sigma$ must hold by Lemma 5.3. Thus, the control parameters have the same value in this case well. This correspondence for the other parameters follows the same reasoning. Thus, the evaluation of the control parameters is equivalent in the two semantics.
Now, we assume first that \( i \) is assigned in \( s_t \) and that \( i \) is not assigned in \( s_f \) nor is an \textbf{abort} in an active position in \( s_f \). Thus, the denotations reduce to the following terms:

\[
C'[\text{if } p \ s_t \ s_f]i\tilde{\kappa}\tilde{\gamma}\sigma = \text{let}
\begin{align*}
\tilde{\kappa}^+ &= \lambda\sigma.\neg\lambda T'(p, \tilde{\gamma}, \sigma) \land \tilde{\kappa}\sigma \land \\
\mathcal{E}[p](\lambda j. \tilde{\gamma}j\sigma), \\
\tilde{\kappa}^- &= \lambda\sigma.\neg\lambda T'(p, \tilde{\gamma}, \sigma) \land \tilde{\kappa}\sigma \land \\
\neg\mathcal{E}[p](\lambda j. \tilde{\gamma}j\sigma), \\
\tilde{\kappa}^T &= \lambda\sigma T'(p, \tilde{\gamma}, \sigma) \land \tilde{\kappa}\sigma \text{ in} \\
(\tilde{\kappa}^T\sigma \rightarrow \oslash, \bot) \sqcup \\
(C'[s_t]i\tilde{\kappa}^+\tilde{\gamma}\sigma) \\
(\tilde{\kappa}^-\sigma \rightarrow \tilde{\gamma}i\sigma, \bot)
\end{align*}
\]

and

\[
\mathcal{P}(\mathcal{G}[\text{if } p \ s_t \ s_f]i\kappa\gamma]\sigma = \mathcal{P}[\text{let}]
\begin{align*}
\kappa^+ &= (T, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}\mathcal{S}(p, \gamma)), \\
\kappa^- &= (F, \kappa, \mathcal{Q}(p, \gamma), p, \mathcal{M}\mathcal{S}(p, \gamma)), \\
\kappa^p &= (\Gamma, \kappa, \mathcal{Q}(p, \gamma), p) \text{ in} \\
(\langle\kappa^p\rangle, \\
(\mathcal{G}[s_t]i\kappa^+\gamma), \\
(\langle\kappa^-, \mathcal{Q}(i, \gamma), i, \{(i, \gamma i)\}\rangle)\|\sigma)
\end{align*}
\]

There are several cases to consider based on the value of the control parameters. If \( \tilde{\kappa}\sigma \) is false, the result for both denotations is \( \bot \) by the induction hypothesis, and since the evaluation of control parameters under the two semantics produce the same results, as shown earlier in this step, if \( \kappa \) is \textbf{false}, then \( \kappa^P, \kappa^+, \text{ and } \kappa^- \) as well as the counterparts \( \tilde{\kappa}, \tilde{\kappa}^T, \tilde{\kappa}^+, \text{ and } \tilde{\kappa}^- \) are all \textbf{false}. If \( \tilde{\kappa}\sigma \) diverges, so do the rest and results are both \( \bot \).

If \( \tilde{\kappa}\sigma \) is \textbf{true}, there are several other cases to consider. If \( \kappa^+\sigma \) is \textbf{true}, \( \kappa^-\sigma \) and \( \kappa^T\sigma \) are both false. The result follows from the induction hypothesis since

\[
C'[\text{if } p \ s_t \ s_f]i\tilde{\kappa}\tilde{\gamma}\sigma = C'[s_t]i\tilde{\kappa}^+\tilde{\gamma}\sigma
\]

and

\[
\mathcal{P}(\mathcal{G}[\text{if } p \ s_t \ s_f]i\kappa\gamma)\sigma = \mathcal{P}[\mathcal{G}[s_t]i\kappa^+\gamma]\|\sigma
\]
If $\bar{\kappa}^-\sigma$ is true, the result follows from the assumptions on the input stores since

$$C'[\text{if } p \ s_t \ s_f] i\bar{\kappa}\bar{\gamma}\sigma = \bar{\kappa}^-\sigma \rightarrow \bar{\gamma}i\sigma, \bot$$
$$= \bar{\gamma}i\sigma$$

and

$$\mathcal{P}(G[\text{if } p \ s_t \ s_f] i\kappa\gamma)\sigma = \mathcal{P}[\langle \kappa^-, Q(i, \gamma), i, \{(i, \gamma i)\} \rangle] \sigma$$
$$= \mathcal{P}[\kappa] \sigma \rightarrow (\mathcal{P}_s[Q(i, \gamma)] \sigma \rightarrow \mathcal{E}[i] \{(i, \mathcal{P}[\gamma i] \sigma), \emptyset\}, \bot)$$
$$= \mathcal{P}[\gamma i] \sigma$$

Finally, if $\bar{\kappa}^T\sigma$ is true, the resulting denotations are

$$C'[\text{if } p \ s_t \ s_f] i\bar{\kappa}\bar{\gamma}\sigma = \bar{\kappa}^T\sigma \rightarrow \emptyset, \bot$$
$$= \emptyset$$

and

$$\mathcal{P}(G[\text{if } p \ s_t \ s_f] i\kappa\gamma)\sigma = \mathcal{P}[\langle \kappa^P \rangle] \sigma$$
$$= \mathcal{P}_k[\langle \kappa^P \rangle] \sigma \rightarrow \emptyset, \bot$$
$$= \emptyset$$

by the correspondence of the control parameters $\kappa^P$ and $\bar{\kappa}^T$.

The other cases follows similarly.

\[\square\]

Now, we prove the lemma establishing the relationship between $T'$ and $Q$.

**Lemma 5.3** For expression $e$ and input stores $\gamma, \bar{\gamma}$ and $\sigma$ such that

$$\forall j. \bar{\gamma}j\sigma = \mathcal{P}[\gamma j] \sigma$$

the following conditions hold:

1) $T'(e, \bar{\gamma}, \sigma)$ iff $\neg \mathcal{P}_s[Q(e, \gamma)] \sigma$

2) $T'(e, \gamma, \sigma)$ diverges iff $\mathcal{P}_s[Q(e, \gamma)] \sigma$ diverges.
Proof  By definition,

\[ T'(e, \bar{\gamma}, \sigma) = \emptyset \in \{ v \mid v = \bar{\gamma}j\sigma \land j \in \mathcal{R}(e) \} \]

and

\[ \mathcal{Q}(e, \gamma) = [k_1, \ldots, k_i] \text{ for } k_i \in \{ \mathcal{F}(k) \mid \langle k \rangle \text{ in } \gamma j \land j \in \mathcal{R}(e) \land k \in p\text{-node} \} \]

First, if \( \mathcal{R}(e) \) is \( \emptyset \), the result follows immediately. For the if direction of Part (1), assume \( T'(e, \bar{\gamma}, \sigma) \) is true. Thus, by the definition of \( T' \), there exists a \( j \in \mathcal{R}(e) \) such that \( \bar{\gamma}j\sigma = \emptyset \). By the assumptions on the stores, \( \mathcal{P}[\bar{\gamma}j\sigma] = \emptyset \). We proceed by examining the cases of \( \gamma j \).

For \( \gamma j = \langle c_k, j \rangle \), the result is immediate since initial stores can not contain abort values. Thus, this case is impossible. For \( \gamma j = \langle c_k \rangle \), the expansion is as follows:

\[ \mathcal{P}[\langle c_k \rangle]\sigma = \mathcal{P}_k[c_k]\sigma \rightarrow \emptyset, \bot \]

For the result to be \( \emptyset \), \( \mathcal{P}_k[c_k]\sigma \) must be true. Therefore, \( \neg \mathcal{P}_a[c_k]\sigma \) must be true since \( \mathcal{P}_k[\mathcal{F}(c_k)]\sigma \) is false.

For \( \gamma j = \langle c_k, c_a, e, c_c \rangle \) where \( c_a = \mathcal{Q}(e, \gamma) \), the expansion is as follows:

\[ \mathcal{P}[\langle c_k, c_a, e, c_c \rangle]\sigma = \mathcal{P}_k[c_k]\sigma \rightarrow (\mathcal{P}_a[c_a]\sigma \rightarrow \mathcal{E}[e]\mathcal{R}\mathcal{S}(c_e, \sigma), \emptyset), \bot \]

Since expression evaluation does not return the abort value, \( \mathcal{P}_a[c_a]\sigma \) must be false, so \( \neg \mathcal{P}_a[c_a]\sigma \) is true. The other cases follow similarly.

For the only if direction of Part (1), assume \( \neg \mathcal{P}_a[\mathcal{Q}(e, \gamma)]\sigma \) holds. By the definition of \( \mathcal{Q} \) and \( \mathcal{P}_a \), there exists a control node \( \kappa_j \) where \( \mathcal{F}(\kappa_j) \in \mathcal{Q}(e, \gamma) \), \( \mathcal{P}_k[\kappa_j]\sigma \) is false, and \( \langle \kappa_j \rangle \) is an abort node the code tree for some \( j \in \mathcal{R}(e) \). By the definition of \( \mathcal{F} \), \( \mathcal{P}_k[\kappa_j]\sigma \) is true and thus, \( \mathcal{P}[\bar{\gamma}j\sigma] = \emptyset \). By the assumptions on the stores, \( \bar{\gamma}j\sigma = \emptyset \) and thus, \( T'(e, \bar{\gamma}, \sigma) \) is true.

Part (2) of the theorem holds since any expression evaluation occurring in the evaluation of the \( T' \) predicate occurs in the evaluation of \( \mathcal{P}_a \) and vice versa.  \( \square \)
5.4.1 Example

To better understand the pdt with abort information, we describe the pdt, shown in Figure 5.11 for the following program:

\[
\text{if } p \quad x := 42 \quad \text{abort;}
\]
\[
z := f(x)
\]

with \(z\) as the demanded identifier. The demand for \(z\) generates a demand for \(x\) which then propagates to the if statement. Since the true branch of the if statement assigns a value for \(x\), the demand is satisfied here. Both branches receive the demand as there is an assignment for \(x\) on the true branch and an abort statement on the false branch. The control dependences for the abort node and the assignment nodes are as expected. The node constructed for the assignment to \(z\), however, includes a control dependence on \(p\) since there is an abort node with that control predecessor in the code table for the expression \(f(x)\).

![Diagram of the example abort program](image)

**Figure 5.11** Example abort Program under Control Perspective
5.5 Alternative View Of Control

The above derivation presents a control perspective on `abort`. The term control perspective comes from the encoding of the abort information in control nodes in the `pdt` that are interpreted before evaluating the expression in the node. This encoding translates abort information into control dependences. Figures 5.12, 5.11 and 5.13 present different `pdgs` for the following program:

\[
\text{if } p \ x := 42 \ \text{abort;}
\]
\[
z := f(x)
\]

Figure 5.12 presents the `pdg` under traditional treatments of `abort`. Figure 5.11 presents the `pdt` specified by the `\mathcal{G}` function in the previous section. In both of these structures, there is a control dependence from the predicate node to the assignment node for `z`. This dependence occurs since the `abort` within the `if` statement potentially causes the assignment statement to abort as well. While the `pdt` in Figure 5.11 is consistent with the literature, it is not a natural derivation from the earlier denotational semantics. The initial semantics characterized `abort` as a data-flow property, in that the abort signal passes with the data values. The derivation above corresponds to the logical extension of the notion of control dependence to the lazy `abort` construct. In this section, we present a data-flow interpretation of `abort` in the final stage of the derivation; the other stages of the derivation are unchanged. The `pdt` derived in this manner does not have additional control dependences as a result of `abort` statements. Instead, an `abort` node is added to the `pdt`; when enabled for evaluation, the `abort` node behaves just as the `abort` statement in the prior semantics — an abort value propagates to all nodes that are data dependent on the `abort` node. While these nodes are formed in the previous derivation, they serve in a control-flow role as opposed to a data-flow role. The final stage for this formulation of the `pdt` appears in Figures 5.15 and 5.16 respectively with the domains given in Figure 5.14. Figure 5.13 contains the `pdt` resulting from the new semantic functions.

The difference in the `\mathcal{G}` function presented here is in the removal of the abort information from the predicate nodes. The data node in Figure 5.15 is the same as that given for language `\mathcal{W}`. The abort information is carried in the data values used to compute expression values. Thus, the partial store for an expression is sufficient to determine whether the abort value is the result of the statement. The predicate nodes are also the same as in language `\mathcal{W}`. The `\Gamma` nodes contain the `abort` information for the predicate expression in the partial store required to computed the value of the
**Figure 5.12** Example abort Program under
Traditional Dependence Definitions

**Figure 5.13** Example abort Program under Data Perspective
Semantic Domains:

\[ \begin{align*}
  w & \in val = val \\
  v & \in aval^T = (val \oplus \otimes) \cup T \\
  i, x & \in ide \\
  e, p & \in exp \\
  \sigma & \in store = ide \rightarrow val^T \\
  \gamma, c_e & \in code-tbl = ide \rightarrow code \\
  c & \in code = d-node \oplus (d-node \otimes d-node \otimes d-node) \\
  & \oplus d-node^+ \\
  c_d & \in d-node = (p-node \otimes exp \otimes code-tbl) \oplus (p-node \otimes ide) \oplus \\
  & (ide \otimes code) \oplus (p-node) \\
  \kappa, c_\kappa & \in p-node = true_c \oplus (p-node \otimes exp \otimes code-tbl) \oplus \\
  & (p-node \otimes exp \otimes code-tbl) \oplus \\
  & (p-node \otimes exp \otimes code-tbl)
\end{align*} \]

Semantics:

\[ \begin{align*}
  G : update \rightarrow stmt \rightarrow ide \rightarrow p-node \rightarrow code-tbl \rightarrow code \\
  E : exp \rightarrow store^T \rightarrow val^T & \quad \text{(abstract)} \\
  P : code \rightarrow store^T \rightarrow aval^T \\
  P_b : p-node \rightarrow store^T \rightarrow bool^T \\
  P_a : code-tbl \rightarrow bool^T
\end{align*} \]

**Figure 5.14** Domains for Data Flow Interpretation of Abort
\[ A_G : \quad G[x := e] = \lambda i. \gamma. i \equiv x \rightarrow (\gamma, e, MS(e, \gamma)), \gamma i \]

\[ I_G : \quad G[\text{if } p \ s_i \ s_f] = \lambda i. \gamma. \neg (A(s_i, i) \lor A(s_f, i)) \rightarrow \gamma i, \]

let

\[ \kappa^+ = (T, \kappa, p, MS(p, \gamma)), \]
\[ \kappa^- = (F, \kappa, p, MS(p, \gamma)), \]
\[ \kappa^P = (T, \kappa, p, MS(p, \gamma)) \text{ in} \]

\[ \langle \kappa^P \rangle, \]
\[ \langle (\neg A(s_i, i) \land \neg AB(s_i, i)) \rightarrow \kappa^+, i, \{(i, \gamma i)\} \rangle, G[s_i][i] \kappa^+ \gamma, \]
\[ \langle (\neg A(s_f, i) \land \neg AB(s_f, i)) \rightarrow \kappa^-, i, \{(i, \gamma i)\} \rangle, G[s_f][i] \kappa^- \gamma \rangle \]

\[ A_G : \quad G[\text{abort}] = \lambda i. \gamma. \langle \kappa \rangle \]

\[ C_G : \quad G[s_1; s_2] = \lambda i. \gamma. G[s_2][i] \kappa (\lambda j. G[s_1][j] \kappa) \gamma \]

\[ W_G : \quad G[\text{while } p \ s] = \lambda i. \gamma. i \notin A(s, i) \rightarrow \gamma i, \]

let \( f = \text{fix}(\lambda \omega. \lambda i. \gamma. \langle \kappa \rangle) \).

let

\[ \kappa^+ = (T, \kappa, p, MS(p, \gamma) \cup MS(i, \gamma)), \]
\[ \kappa^- = (F, \kappa, p, MS(p, \gamma) \cup MS(i, \gamma)), \]
\[ \kappa^P = (T, \kappa, p, MS(p, \gamma) \cup MS(i, \gamma)) \text{ in} \]

\[ \langle \kappa^-, i, \{(i, \gamma i)\} \rangle \circ (\kappa^P) \circ \text{wfix}^+(\lambda j. G[s][j] \kappa^+ \gamma) \rangle \text{ in} \]

let

\[ \kappa^+ = (T, \kappa, p, MS(p, \gamma)), \]
\[ \kappa^- = (F, \kappa, p, MS(p, \gamma)), \]
\[ \kappa^P = (T, \kappa, p, MS(p, \gamma)) \text{ in} \]

\[ \langle \kappa^+, i, \{(i, \gamma i)\} \rangle \circ (\kappa^P) \circ ffix^+(\lambda j. G[s][j] \kappa^+ \gamma) \rangle \]

\[ F_G : \quad G[s; \text{end}] = \lambda i. \gamma. G[s][i] \kappa \gamma \]

where

\[ MS(e, \gamma) = \{(j, \gamma j) \mid j \in R(e)\} \]

\textbf{Figure 5.15} Data Flow Interpretation of Abort
predicate expression. The code for while loops includes in the store for the rest of the loop iterations the value of the demanded identifier from the prior iteration. Since the store is used to signal the abort, the store for the predicate expression is combined with the store for the demanded identifier to obtain the abort information for the loop predicate. The other nodes are unchanged.

The P and Pa functions are unchanged except for using the expression store as the argument to the Pa function to verify the abort status. The function Pa returns false if any of the expression values in the store are the abort value, and true otherwise.

As in the alternate semantics, the initial code tree argument creates the iden nodes for identifiers that are used before being assigned in the program. With this initial code-tree parameter, the functions G, P and C' satisfy the invariant specified in the following theorem.

**Theorem 5.6** For all programs P, identifiers i, and stores σ,

\[ P(G[P]i\text{true}_e\gamma_0)σ = C'[P]i(λσ.T)(λiσ.σj)σ \]

---

\[ P[\{c_k, e, c_e\}] = \lambdaσ.\text{P}_k[\text{c}_k]σ → (\text{P}_a[\text{c}_e]σ → E[e]RS(c_e, σ), ∅), ⊥ \]

\[ P[\{c_k, i\}] = \lambdaσ.\text{P}_k[\text{c}_k]σ → σi, ⊥ \]

\[ P[\{i, c_i\}] = \lambdaσ.\text{P}[c_i]σ \]

\[ P[\{c_k\}] = \lambdaσ.\text{P}_k[\text{c}_k]σ → ∅, ⊥ \]

\[ P[\{c_a, c_i, c_f\}] = \lambdaσ.\text{P}[c_a]σ \cup \text{P}[c_i]σ \cup \text{P}[c_f]σ \]

\[ P[\{c_1, c_2, \ldots\}] = \lambdaσ.\text{P}[c_1]σ \cup \text{P}[c_2, \ldots]σ \]

---

\[ \text{P}_k[\{\text{true}_c\}] = \lambdaσ.T \]

\[ \text{P}_k[\{T, c_k, p, c_p\}] = \lambdaσ.\text{P}_a[\text{c}_p]σ → \text{P}_k[\text{c}_k]σ ∧ E[p]RS(c_p, σ), F \]

\[ \text{P}_k[\{F, c_k, p, c_p\}] = \lambdaσ.\text{P}_a[\text{c}_p]σ → \text{P}_k[\text{c}_k]σ ∧ ¬E[p]RS(c_p, σ), F \]

\[ \text{P}_k[\{T, c_k, p, c_p\}] = \lambdaσ.\text{P}_k[\text{c}_k]σ ∧ ¬\text{P}_a[\text{c}_p]σ \]

\[ \text{P}_a[\{\}]] = \lambdaσ.T \]

\[ \text{P}_a[\{(j, c_j)\}] = \lambdaσ.\text{P}[c_j]σ \Rightarrow ∅ → F, T \]

\[ \text{P}_a[\{(j, c_j), (k, c_k), \ldots\}] = \lambdaσ.\text{P}_a[\{(j, c_j)\}]σ ∧ \text{P}_a[\{(k, c_k), \ldots\}]σ \]

\[ RS(C_e, σ) = \{(b, P[C_b]σ) \mid (b, C_b) ∈ C_e \} \]

**Figure 5.16** Data Flow Interpretation of Abort (cont)
**Proof**  To prove the correspondence, we prove a stronger lemma first. Since the initial parameters satisfy the requirements of the lemma, the theorem follows from the lemma.

**Lemma 5.4**  Let $P$ be a statement, $\sigma$ a store, $\gamma$ and $\bar{\gamma}$ code trees, $i$ an identifier, and $\kappa$ and $\bar{\kappa}$ control parameters. If

1) $\forall j. \gamma j \sigma = \mathcal{P}[\gamma j] \sigma$

2) $\bar{\kappa} \sigma = \mathcal{P}[\kappa] \sigma$

then

$$\mathcal{P}(G[P]i\kappa \gamma) \sigma = C'[P]i\bar{\kappa} \bar{\gamma} \sigma$$

**Proof**  The proof proceeds by induction on the structure of $P$ as in the previous section.

**Case** $x := e$  There are two cases to consider. If $i \not= x$, the applicable denotations are

$$\mathcal{P}(G[x := e]i\kappa \gamma) \sigma = \gamma i$$

and

$$C'[P]i\bar{\kappa} \bar{\gamma} \sigma = \bar{\gamma} i \sigma$$

By the assumptions on the inputs, these two terms are equal.

If $i = x$, the applicable denotations are as follows:

\[
\mathcal{P}[G[P]i\kappa \gamma] \sigma = \mathcal{P}[\kappa, \{(j, \gamma j) \mid j \in \mathcal{R}(e)\}] \sigma \\
= \mathcal{P}[\kappa] \sigma \rightarrow \\
(\mathcal{P}[\mathcal{E}([j, P[\gamma j] \sigma] \mid j \in \mathcal{R}(e)]) \sigma \rightarrow \\
\mathcal{E}[e]((j, P[\gamma j] \sigma) \mid j \in \mathcal{R}(e)), \\
\mathcal{E}[e]((\lambda j. \bar{\gamma} j) \sigma), \bot)
\]

and

$$C'[P]i\bar{\kappa} \bar{\gamma} \sigma = \bar{\kappa} \sigma \rightarrow (T'(e, \bar{\gamma}, \sigma) \rightarrow \bot, \mathcal{E}[e]((\lambda j. \bar{\gamma} j) \sigma), \bot)$$

There are several cases to consider here. If $\bar{\kappa} \sigma$ diverges, by the assumptions on the inputs, $\mathcal{P}[\kappa] \sigma$ also diverges. If $\bar{\kappa} \sigma$ is false, so is $\mathcal{P}[\kappa] \sigma$ by the assumptions on the inputs, and both denotations are
therefore $\emptyset$. If $\kappa \sigma$ is true, so is $\mathcal{P}_\kappa \llbracket \kappa \rrbracket \sigma$ by the assumptions on the inputs. Thus, the denotations reduce to the following:

$$
\mathcal{P}(\mathcal{G} \llbracket P \rrbracket i \kappa \gamma) \sigma = \mathcal{P}_\kappa \llbracket \{(j, \gamma j) \mid j \in \mathcal{R}(e)\} \rrbracket \sigma \rightarrow \\
\mathcal{E}[e] \llbracket \{(j, P \llbracket \gamma j \rrbracket \sigma) \mid j \in \mathcal{R}(e)\}, \emptyset
$$

and

$$
\mathcal{C}'[P] i \kappa \gamma \sigma = T'(e, \bar{\gamma}, \sigma) \rightarrow \emptyset, \mathcal{E}[e]((\lambda j. \bar{\gamma} j) \sigma)
$$

Assume first that $T'(e, \bar{\gamma})$ is true. By Lemma 5.5 shown below, $\mathcal{P}_\kappa \llbracket \{(j, \gamma j) \mid j \in \mathcal{R}(e)\} \rrbracket \sigma$ is false. Thus, both denotations are $\emptyset$. If instead we assume that $T'(e, \bar{\gamma})$ is false, $\mathcal{P}_\kappa \llbracket \{(j, \gamma j) \mid j \in \mathcal{R}(e)\} \rrbracket \sigma$ is true also by Lemma 5.5. The denotations are therefore as follows:

$$
\mathcal{P}(\mathcal{G} \llbracket P \rrbracket i \kappa \gamma) \sigma = \mathcal{E}[e] \llbracket \{(j, P \llbracket \gamma j \rrbracket \sigma) \mid j \in \mathcal{R}(e)\}
$$

and

$$
\mathcal{C}'[P] i \kappa \gamma \sigma = \mathcal{E}[e]((\lambda j. \bar{\gamma} j) \sigma)
$$

These expressions are equal by the assumptions on the inputs and the fact that expression evaluation depends only on the values in the store for identifiers referenced in the expression. The theorem therefore holds for assignment statements.

Case $s_1; s_2$ By the assumptions on the inputs and the induction hypothesis,

$$
\forall j. \mathcal{C}'[s_1] j \kappa \gamma \sigma = \mathcal{P}[\mathcal{G}[s_1] j \kappa \gamma] \sigma
$$

Therefore, the inputs to the denotations

$$
\mathcal{C}'[s_1; s_2] i \kappa \gamma \sigma = \mathcal{C}'[s_2] i \kappa(\lambda j. \mathcal{C}'[s_1] j \kappa \gamma)
$$

and

$$
\mathcal{P}[\mathcal{G}[s_1; s_2] i \kappa \gamma] \sigma = \mathcal{P}(\mathcal{G}[s_2] i \kappa(\lambda j. \mathcal{G}[s_1] j \kappa \gamma)) \sigma
$$

satisfy the assumptions of the theorem. Thus, the induction hypothesis applies for statement $s_2$ and the theorem holds for statement composition.

The second part of the theorem follows directly from the induction hypothesis.
Case if $p$, $s_i$, $s_f$. As with the assignment clause, there are two cases to consider. First, if the identifier $i$ is not assigned in the body of the if statement, the denotations are

$$C'[P][i\tilde{\kappa}\tilde{\gamma}\sigma] = \gamma i\sigma$$

and

$$\mathcal{P}[G[P][i\kappa\gamma]\sigma] = \mathcal{P}[\gamma i]\sigma$$

These terms are equal by the assumptions on the input stores. If the identifier is assigned within the body of the if, there are several cases to consider. First, consider the control parameters in the two denotations, using $\kappa^+$ and $\bar{\kappa}^+$ as examples:

$$\bar{\kappa}^+\sigma = -T'(p, \tilde{\gamma}, \sigma) \land \bar{\kappa}\sigma \land \mathcal{E}[p](\lambda j.\tilde{\gamma}j\sigma)$$

and

$$\mathcal{P}_k[\kappa^+]\sigma = \mathcal{P}_k[(T, \kappa, p, \{ (j, \gamma j) \mid j \in \mathcal{R}(p) \})\sigma \rightarrow \mathcal{P}_k[\kappa]\sigma \land \mathcal{E}[p](\{ (j, \mathcal{P}[\gamma j]) \mid j \in \mathcal{R}(p) \}), F]$$

By the assumptions on the input stores and the properties of expression evaluation,

$$\mathcal{E}[p](\lambda j.\tilde{\gamma}j\sigma) = \mathcal{E}[p](\{ (j, \mathcal{P}[\gamma j]) \mid j \in \mathcal{R}(p) \})$$

and by the assumptions on the input control parameters,

$$\bar{\kappa}\sigma = \mathcal{P}_k[\kappa]\sigma$$

Finally, if $T'(p, \tilde{\gamma}, \sigma)$ is true, $-\mathcal{P}_a[\{ (j, \gamma j) \mid j \in \mathcal{R}(p) \}]\sigma$ must also hold by Lemma 5.5, and thus both control parameters are false. By Lemma 5.5, if one expression diverges, so does the other. If $T'(p, \tilde{\gamma}, \sigma)$ is false, $\mathcal{P}_a[\{ (j, \gamma j) \mid j \in \mathcal{R}(p) \}]\sigma$ must also hold by Lemma 5.5. Thus, the control parameters have the same value in this case well. This correspondence for the other parameters follows using the same reasoning. Thus, the evaluation of the control parameters is equivalent in the two semantics.
Now, we assume first that \( i \) is assigned in \( s_t \) and that \( i \) is not assigned in \( s_f \) nor is an abort in an active position in \( s_f \). Thus, the denotations reduce to the following terms:

\[
C'[\text{if } p \ s_t \ s_f][i\bar{\kappa}\bar{\gamma}\sigma] = \text{let}\\
\bar{\kappa}^+ = \lambda \sigma \neg T'(p, \bar{\gamma}, \sigma) \land \bar{\kappa}\sigma \land \\
E[p](\lambda j.\bar{\gamma}j\sigma), \\
\bar{\kappa}^- = \lambda \sigma.\neg T'(p, \bar{\gamma}, \sigma) \land \bar{\kappa}\sigma \land \\
\neg E[p](\lambda j.\bar{\gamma}j\sigma), \\
\bar{\kappa}^T = \lambda \sigma.T'(p, \bar{\gamma}, \sigma) \land \bar{\kappa}\sigma \text{ in} \\
(\bar{\kappa}^T\sigma \rightarrow \emptyset, \bot) \cup \\
(C'[s_t][i\bar{\kappa}^+\bar{\gamma}\sigma] \\
(\bar{\kappa}^-\sigma \rightarrow \bar{\gamma}i\sigma, \bot)
\]

and

\[
P[\mathcal{G}[\text{if } p \ s_t \ s_f][i\kappa\gamma]\sigma] = P[\text{let}\\n\kappa^+ = \langle T, \kappa, p, \mathcal{M}(p, \gamma) \rangle, \\
\kappa^- = \langle F, \kappa, p, \mathcal{M}(p, \gamma) \rangle, \\
\kappa^P = \langle G, \kappa, p, \mathcal{M}(p, \gamma) \rangle \text{ in} \\
(\langle \kappa^P \rangle, \\
(\mathcal{G}[s_t][i\kappa^+\gamma], \\
(\langle \kappa^- \rangle, i, \{(i, \gamma_i)\}))\sigma
\]

There are several cases to consider based on the value of the control parameters. If \( \bar{\kappa}\sigma \) is \text{false}, the result for both denotations is \( \bot \) by the induction hypothesis, the correspondence of control parameters between the two semantics, and since if \( \kappa \) is \text{false}, then \( \kappa^P \), \( \kappa^+ \), and \( \kappa^- \) as well as the counterparts \( \bar{\kappa}, \bar{\kappa}^T, \bar{\kappa}^+ \), and \( \bar{\kappa}^- \) are all \text{false}. If \( \bar{\kappa}\sigma \) diverges, so do the rest and both denotations are \( \bot \).

If \( \bar{\kappa}\sigma \) is \text{true}, there are several other cases to consider. If \( \bar{\kappa}^+\sigma \) is \text{true}, \( \bar{\kappa}^-\sigma \) and \( \bar{\kappa}^T\sigma \) are both \text{false}. The result follows from the induction hypothesis and Lemma 5.1 since

\[
C'[\text{if } p \ s_t \ s_f][i\bar{\kappa}\bar{\gamma}\sigma] = C'[s_t][i\bar{\kappa}^+\bar{\gamma}\sigma]
\]

and

\[
P[\mathcal{G}[\text{if } p \ s_t \ s_f][i\kappa\gamma]\sigma] = P[\mathcal{G}[s_t][i\kappa^+\gamma]\sigma]
\]
If $\bar{\kappa}^- \sigma$ is true, the result follows from the assumptions on the input stores and Lemma 5.5 since

$$C'[\text{if } p \text{ s}_t \text{ s}_f]i\bar{\kappa}\bar{\gamma}\sigma = \bar{\kappa}^- \sigma \rightarrow \bar{\gamma}i\sigma, \bot$$

$$= \bar{\gamma}i\sigma$$

and

$$\mathcal{P}[\mathcal{G}[\text{if } p \text{ s}_t \text{ s}_f]i\kappa\gamma] \sigma = \mathcal{P}[(\kappa^-, i, \{(i, \gamma_i)\})] \sigma$$

$$= \mathcal{P}[\kappa] \sigma \rightarrow (\mathcal{P}_a[\{(i, \gamma_i)\}] \rightarrow$$

$$\mathcal{E}[i]\{(i, \mathcal{P}[\gamma_i] \sigma)\}, \emptyset), \bot$$

$$= \mathcal{P}[\gamma_i] \sigma$$

Finally, if $\bar{\kappa}^T \sigma$ is true, the resulting denotations are

$$C'[\text{if } p \text{ s}_t \text{ s}_f]i\bar{\kappa}\bar{\gamma}\sigma = \bar{\kappa}^T \sigma \rightarrow \emptyset, \bot$$

$$= \emptyset$$

and

$$\mathcal{P}[\mathcal{G}[\text{if } p \text{ s}_t \text{ s}_f]i\kappa\gamma] \sigma = \mathcal{P}[(\kappa^P)] \sigma$$

$$= \mathcal{P}_k[\kappa^P] \sigma \rightarrow \emptyset, \bot$$

$$= \emptyset$$

by the correspondence of the control parameters $\kappa^P$ and $\bar{\kappa}^T$.

The other cases follows similarly.

\[\square\]

Now, we prove the lemma establishing the relationship between $T'$ and $\mathcal{P}_a$.

**Lemma 5.5** For expression $e$ and input stores $\gamma$, $\bar{\gamma}$ and $\sigma$ such that

$$\forall j. \bar{\gamma}j \sigma = \mathcal{P}[\gamma_j] \sigma$$

the following condition holds:

$$\neg \mathcal{P}_a[\mathcal{M}S(e, \gamma)] \sigma = T'(e, \bar{\gamma}, \sigma)$$
Proof By definition,
\[ T'(e, \tilde{\gamma}, \sigma) = \emptyset \in \{ v \mid v = \tilde{\gamma}j\sigma \land j \in \mathcal{R}(e) \} \]
and
\[ \mathcal{MS}(e, \gamma) = \{ (j, \gamma j) \mid j \in \mathcal{R}(e) \} \]

The result follows directly from the definition of \( P_a \) and the assumptions on the inputs.

5.6 Control Dependence in the Presence of Abort

With the addition of an abort construct, the control dependence relation under the control interpretation no longer reflects the nesting structure of the program. Predicates governing abort statements are control dependence predecessors of statements outside the syntactic scope of the predicate. The lazy semantics assigned to the abort construct in this chapter is different from the semantics generally used for abort. However, the lazy interpretation provides greater opportunities for parallelism than the traditional interpretation since there are fewer dependences using the lazy interpretation.

With the control interpretation of abort, statements can have multiple control dependence predecessors. However, unlike the multiple predecessors present in programs exploiting non-local control constructs, any mismatch of a predicate disallows the evaluation of the dependent statement. Instead, the multiple control predecessors arise from either multiple abort statements potentially influencing a statement, or a single abort statement for a statement within a structured control construct. In the general case, a mismatch means the statement may not execute; in the abort case, a mismatch requires that the statement must not execute. For example, the assignment node for \( z \) in the following program, whose \( pdt \) appears in Figure 5.17, is control dependent on both predicate nodes:

\[
\text{if } p_1 \text{ abort } x := 1; \text{ if } p_2 \text{ abort } y := 2; z := x + y
\]

If either of the predicates is true, this assignment is aborted.

Under the data interpretation, there are no additional control dependences, and the invariant that each node has at most one immediate control dependence predecessor still holds. The abort information is carried in the flow edges from the abort
nodes and the `abort` control nodes. The data dependence relations are unchanged by the addition of `abort` except for these additional flow edges. As in the array chapter, the lazy semantics is presented throughout this chapter. The use of the lackadaisical semantics introduces additional control dependences due to the propagation of demand for the assigned identifier in addition to the referenced identifier. The derivation remains the same, however.

5.7 Operational Semantics for PDTs with Abort

Since there are two formulations of `pdts` with `abort`, this section presents two modifications to the operational semantics of language $\mathcal{W}$.

5.7.1 Data Interpretation of Abort

The domains for `pdt graphs` under the data interpretation of `abort` differ from those for language $\mathcal{W}$ in the addition of the `abort` node and the `abort` control node. The interpretation of the other tree components as nodes and edges remains unchanged.

Figure 5.17 Example for Control Dependence and `abort`
For the abort node, a data flow edge exists from an abort node to its successor. There is an incoming control dependence from the node for its control component. For the abort control node, there is also a data flow edge to its successor. This node allows the propagation of the abort token when it is a predicate expression that encounters the abort. The node definitions are given in Figure 5.18.

The abort control node functions like a data node in that a value is propagated as a result of the rewriting if the predicate expression encounters an abort value. Thus, the abort control node is considered a data node rather than a predicate node. The rewriting rules for the other nodes verify that no abort values are present in the expressions before evaluating the expression. These changes and additions are reflected in the rewriting rules presented in Figure 5.19.

The predicate $L$ used in the rewriting rules for the assignment nodes and the predicate nodes tests the expression to determine if any of the values propagated for that expression were abort values. This test is analogous to the test $T$ that is performed in the denotational semantics, testing for the presence of an abort value. If the abort value is present, the abort value is propagated to the successors of the nodes.

### 5.7.2 Control Interpretation of Abort

As in the the previous section, new nodes are introduced into the pdt graph with the abort construct. However, in the control interpretation of abort, the interpretation of the new nodes is different. The abort node and the abort control nodes add

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>idef</code></td>
<td>$(i_m, x)$</td>
</tr>
<tr>
<td><code>set</code></td>
<td>$(s_m, x := e)$</td>
</tr>
<tr>
<td><code>abort</code></td>
<td>$(s_m, x)$</td>
</tr>
<tr>
<td><code>start</code></td>
<td>$(p^s_m, \text{true}_c)$</td>
</tr>
<tr>
<td><code>true</code></td>
<td>$(p^t_m, e)$</td>
</tr>
<tr>
<td><code>false</code></td>
<td>$(p^f_m, e)$</td>
</tr>
<tr>
<td><code>cabort</code></td>
<td>$(p^c_m, x := e)$</td>
</tr>
<tr>
<td><code>final</code></td>
<td>$(f_m, x)$</td>
</tr>
</tbody>
</table>

**Figure 5.18** Representations for PDG Nodes
\begin{align*}
\langle i_r, x \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_I = N_I - \{r\} \\
N'_k = N_k[n \leftarrow x/\sigma x] \text{ for } k \in \{S, P, F\}
\end{array} \right. \\
\langle s_r, x := e \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_I = N_I \\
N'_S = (N_S - \{r\})[n \leftarrow x/v] \\
N'_k = N_k[n \leftarrow x/v] \text{ for } k \in \{P, F\}
\end{array} \right. \\
\text{where} & \\
v = \left\{ \begin{array}{ll}
\mathcal{E}[e] & \text{if } \mathcal{L}(e) \text{ is true} \\
\sigma & \text{otherwise}
\end{array} \right. \\
\text{and} & \\
\mathcal{L}(e) \text{ is true} & \text{iff } e \text{ contains no } v = \sigma
\end{align*}

\begin{align*}
\langle e_r, x := e \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_I = N_I \\
N'_S = (N_S - \{r\})[n \leftarrow x/\sigma] \\
N'_k = N_k[n \leftarrow x/\sigma] \text{ for } k \in \{P, F\}
\end{array} \right. \\
\langle p_r^b, e \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_I = N_k - Q_k \text{ for } k \in \{I, S, F\} \\
N'_P = N_P - Q_P - \{r\}
\end{array} \right.
\end{align*}

for

\begin{align*}
Q & = \left\{ \begin{array}{l}
\{m \mid \langle p_r^b, e \rangle \in Q, S(q) = m\} \cup \{n\} & \text{if } \mathcal{E}[e] \neq b \wedge \mathcal{L}(e) = \text{true} \\
\emptyset & \text{otherwise}
\end{array} \right. \\
Q_i & = \{m \mid m \in Q \wedge m \in N_i\} \text{ for } i \in \{I, S, P, F\}
\end{align*}

\begin{align*}
\langle p_r^a, x := e \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_k = N_k \text{ for } k \in \{I, P, F\} \\
N'_S = N_S - \{r\}
\end{array} \right. \\
\text{if} & \\
\mathcal{L}(e) \text{ is true}
\end{align*}

\begin{align*}
\langle p_r^a, x := e \rangle, S(r) = n \quad & \left\{ \begin{array}{l}
N'_I = N_I \\
N'_S = (N_S - \{r\})[n \leftarrow x/\sigma] \\
N'_k = N_k[n \leftarrow x/\sigma] \text{ for } k \in \{P, F\}
\end{array} \right. \\
\text{if} & \\
\mathcal{L}(e) \text{ is false}
\end{align*}

\textbf{Figure 5.19} Rewriting Rules for Data Abort
\[(i, x), S(r) = n \quad \begin{aligned} N'_I &= N_I \setminus \{r\} \\
'_k &= N_k[n \leftarrow x/\sigma x] \text{ for } k \in \{S, P, F\} \end{aligned} \]

\[(s, x := e), S(r) = n \quad \begin{aligned} N'_I &= N_I \\
'_S &= (N_S \setminus \{r\})[n \leftarrow \mathcal{E}[e]] \\
'_k &= N_k[n \leftarrow \mathcal{E}[e]] \text{ for } k \in \{P, F\} \end{aligned} \]

\[(e, x := e), S(r) = n \quad \begin{aligned} N'_k &= N_k \text{ for } k \in \{I, P, F\} \\
'_S &= N_S \setminus \{r\} \end{aligned} \]

\[(p, e), S(r) = n \quad \begin{aligned} N'_k &= N_k - Q_I \text{ for } k \in \{I, S, F\} \\
'_p &= N_p - Q_P \setminus \{r\} \end{aligned} \]

for \(b \in \{t, f\}\)

where \(Q = \left\{ \begin{array}{ll} \emptyset & \text{if } \mathcal{E}[e] = b \lor e \equiv \text{true}_c \\ \{ m \mid (p^b_{<i'}, e') \in Q, S(q) = m \} \cup \{ n \} & \text{if } \mathcal{E}[e] \neq b \end{array} \right\}\)

\(Q_i = \{ m \mid m \in Q \land m \in N_i \} \text{ for } i \in \{I, S, P, F\}\)

\[(p^e_{<i}, x := e), S(r) = n \quad \begin{aligned} N'_k &= N_k \text{ for } k \in \{I, P, F\} \\
'_S &= N_S \setminus \{r\} \end{aligned} \]

\textbf{Figure 5.20} Rewriting Rules for Control Abort
no information to the control \textit{pdt} since the abort information is now represented as additional control dependence edges. The following example illustrates the differences between the two interpretations:

\begin{verbatim}
if p abort  x := 3;
y := f(x);
z := f(y)
\end{verbatim}

The \textit{pdt} under the data interpretation for this program fragment, with a demand for the identifier \texttt{z}, appears in Figure 5.21. In this example, the \texttt{abort} node has a data flow edge to the assignment node for \texttt{y}, but there is no data flow edge from the \texttt{abort} node to the assignment node for \texttt{z}. The only control dependence edges are from the predicate node \texttt{p} to the \texttt{abort} node, labeled \texttt{T}, and to the assignment node for \texttt{x}, labeled \texttt{F}. If the predicate \texttt{p} is \texttt{true}, the abort value propagates over the data flow edge to the assignment node for \texttt{y}. Since the store for the expression contains an abort value, this node propagates the abort value to the assignment node for \texttt{z}.

The \textit{pdt}, appearing in Figure 5.22, using the control approach includes additional control dependences. This \textit{pdt} includes control dependences labeled \texttt{F} from the pred-
icate \( p \) to the assignments for both \( y \) and \( z \). In this case, if the predicate \( p \) is \texttt{true}, the assignments for \( y \) and \( z \), as well as \( x \), denote \( \bot \) since their control nodes denote \texttt{false}. From the operational perspective, nodes that under the data interpretation receive an abort value are removed from the graph without being evaluated under the control interpretation.

Data and predicate nodes each have an additional component specifying the \texttt{abort} predicate information for the node. Each predicate tuple in the abort information defines an incoming control dependence edge for the node. The rest of the mapping from the tuple presentation to the graph presentation of the \texttt{pdt} remains the same as that in the previous section. The representations for the nodes in the graph are given in Figure 5.18 in the previous section. Figure 5.20 contains the rewriting rules for the operational semantics for the control interpretation of the \texttt{abort} construct. These rules for the assignment, predicate, and assignment nodes are the same as those given for language \( \mathcal{W} \). The rules for the additional nodes remove the node from the graph with no other effects.

5.8 Discussion

We have presented two different mechanisms for incorporating an \texttt{abort} mechanism into the \texttt{pdt}, the control interpretation and the data interpretation. In the control interpretation, additional control dependence edges encode the effects of the \texttt{abort} construct. Nodes in the \texttt{pdt} may now have multiple incoming control dependences; however, the meaning of a control dependence edge remains the same as in the pre-
vious chapters. The *abort* node and the *abort* control nodes are not required in control interpretation. In the data interpretation, an abort value flows to all nodes affected by an *abort*; the *abort* node is the source of the abort value. The *abort* control node propagates the abort value when it is encountered while evaluating a predicate expression.

The denotational definitions for the two approaches are quite similar although the control approach is more complex. The additional complexity relates to the creation of the additional control dependences. In the operational definitions, however, the control approach is much simpler. The additional control dependence edges are handled using the same rules as the other control dependences. The *abort* nodes are not required to determine the meaning of the *pdt* under the operational semantics. For the data approach, however, the new nodes have to be interpreted, and each expression evaluation must check for the presence of the abort value.

The data interpretation of *abort* follows more naturally from the extended direct semantics for *abort* than the control interpretation does. The control interpretation, however, is inspired by the intuition that an *abort* corresponds to a *goto* statement with the end of the program as the target of the *goto*. The general control mechanism of *goto* is naturally characterized by the notion of control dependence. The data interpretation also allows scoping of the effects of an *abort* statement, allowing a Lisp-style catch and throw mechanism to be easily implemented. For these reasons, both interpretations are described in this dissertation. For the purposes of providing tools for researchers working with the *pdt*, the control interpretation is superior. However, with respect to the flexibility of the approach, the data interpretation is superior.

In the next chapter, the *goto* statement is added to the language. Since an *abort* statement is conceptually like a *goto* to an exit statement, the use of a separate *abort* construct seems redundant. However, in Section 8.1.7 in the final chapter the distinction between *abort* and *goto* exit is discussed. The addition of the *abort* construct allows additional parallelism to be detected, even in the presence of a general control flow construct.
Chapter 6

Semantics of Goto

Many existing programs exploit general control operations. Thus, a usable theory of program dependence must confront the issue of the non-structured control flow which results from the use of these operations. This chapter addresses the addition of the goto statement, one such operation. This addition impacts the control dependence relation, whose definition changes in the presence of such operations. The first section describes the goto statement and its semantics. Section 6.2 presents the basic approach for accommodating general control flow, while Section 6.3 refines this approach to expose more parallelism and to reflect the traditional definition of control dependence. Finally, Section 6.4 describes the operational semantics.

6.1 Language \( \mathcal{W} \) with General Control Flow

The language extension for general control operations allows labels on statements in the program, and adds a goto statement which transfers control to the specified label. The syntax of the extended language includes optional labels on statements as well as the goto statement itself:

\[
\begin{align*}
x & \in \text{id} \\
e & \in \text{exp} \\
l & \in \text{lab} \\
s & ::= \text{if } e \; ls \; ls | \text{while } e \; ls \; | x := e \; ls ; ls | \text{goto } l \\
ls & ::= l : s ; ls | l : s | ls | s \\
P & ::= ls ; \text{end}
\end{align*}
\]

where \( l \) ranges over statement labels, and labels on statements are optional. All labels are at the program level; there is no local scoping of the statement labels. Labels can occur within the branches of an if statement and within the body of a while loop.
6.1.1 Semantics of goto

Denotational semantics for languages supporting goto statements generally utilize continuation-passing style models [54]. Figure 6.1 presents a semantics in this style for the goto language. The meaning function \( \mathcal{M} \) uses a continuation argument, \( k \), to guide the processing. Continuations are functions mapping stores to stores, representing the meaning of the rest of the program. The clause for assignment statements sends the updated store to the continuation \( k \). The if statement clause uses the result of the predicate expression \( p \) to determine whether the true or the false branch is invoked with the continuation of the if statement. The while statement clause uses the continuation as the result if the predicate is false. The goto statement clause extracts the continuation for the label of the target of the goto statement from the label environment and sends the store to this continuation, discarding the continuation \( k \). The clause for statement composition uses as the continuation for statement \( s_1 \) the meaning of statement \( s_2 \) in the continuation \( k \).

The label function \( \mathcal{J} \) defines the continuation associated with each label found in a program and creates a label environment containing this information. The clauses for the assignment and goto statements do not contribute any information to the label environment since these statements have no components that can be labeled. The clause for the if statement, \( \mathcal{L}_J \), combines the environments for the two branches of the if statement using the mkenv function. This help function has the empty function as the identity element and is defined over finite functions represented as finite tables. The while statement clause uses the continuation found for the while loop under the meaning function \( \mathcal{M} \) as the continuation for the body of the loop. The statement composition clause, \( \mathcal{C}_J \) combines the label environments of the component statements. The continuation for statement \( s_2 \) is the continuation for the composition as a whole, \( k \). The continuation for statement \( s_1 \) is the meaning for statement \( s_2 \) as specified by the meaning function \( \mathcal{M} \). Finally, the labeled statement clause, \( \mathcal{L}_J \), creates an entry in the label environment with the label \( l \) mapping to the meaning of statement \( s \). This label function correctly specifies the continuation associated with a label when goto labels are present in the branches of conditional statements and within the body of a loop.\(^{12}\)

\(^{12}\)Gordon [27] presents a more detailed explanation of the label function for this language.
Semantic Domains:
\[
\begin{align*}
P & \in \text{pgm} \\
s, s_i & \in \text{stmt, lstmt} \\
x & \in \text{ide} \\
e, p & \in \text{exp} \\
l & \in \text{lab} \\
\rho & \in \text{labenv} = \text{lab} \rightarrow \text{cont} \\
\sigma & \in \text{store} = \text{ide} \rightarrow \text{val} \\
k & \in \text{cont} = \text{store} \rightarrow \text{store}
\end{align*}
\]

Semantics:
\[
\begin{align*}
\mathcal{M}_p &: \text{pgm} \rightarrow \text{store} \rightarrow \text{store} \\
\mathcal{J} &: \text{lstmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{labenv} \\
\mathcal{M} &: \text{lstmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{cont}
\end{align*}
\]
\[
\mathcal{M}_p[s; \text{end}] = \lambda \sigma.\mathcal{M}[s] \rho(\lambda \sigma.\sigma)\sigma
\]
where \( \rho = \text{fix}(\lambda r.\mathcal{J}[s]r(\lambda \sigma.\sigma)) \)

\[
\begin{align*}
\mathcal{A}_M &: \mathcal{M}[x := e] = \lambda pk.\sigma.\text{send}(k, \delta(\sigma, x, \mathcal{E}[e]_\sigma)) \\
\mathcal{I}_M &: \mathcal{M}[\text{if } p s_i s_f] = \lambda pk.\sigma.\mathcal{E}[p]_\sigma \rightarrow \mathcal{M}[s_i]_\rho k \sigma, \mathcal{M}[s_f]_\rho k \sigma \\
\mathcal{C}_M &: \mathcal{M}[s_1; s_2] = \lambda pk.\sigma.\mathcal{M}[s_1]_\rho(\mathcal{M}[s_2]_\rho k)\sigma \\
\mathcal{W}_M &: \mathcal{M}[\text{while } p s] = \lambda pk.\text{fix}(\lambda f.\lambda \sigma'.\mathcal{E}[p]_\rho \sigma' \rightarrow \mathcal{M}[s]_\rho f \sigma', \text{send}(k, \sigma')) \\
\mathcal{G}_M &: \mathcal{M}[\text{goto } l] = \lambda pk.\sigma.\text{send}(pl, \sigma) \\
\mathcal{L}_M &: \mathcal{M}[l : s] = \lambda pk.\sigma.\mathcal{M}[s]_\rho k \sigma
\end{align*}
\]

\[
\begin{align*}
\mathcal{A}_J &: \mathcal{J}[x := e] = \lambda pk.() \\
\mathcal{I}_J &: \mathcal{J}[\text{if } p s_i s_f] = \lambda pk.\text{mkenv}(\mathcal{J}[s_i]_\rho k, \mathcal{J}[s_f]_\rho k) \\
\mathcal{C}_J &: \mathcal{J}[s_1; s_2] = \lambda pk.\text{mkenv}(\mathcal{J}[s_1]_\rho(\lambda \sigma.\mathcal{M}[s_2]_\rho k \sigma), \mathcal{J}[s_2]_\rho k) \\
\mathcal{W}_J &: \mathcal{J}[\text{while } p s] = \lambda pk.\mathcal{J}[s]_\rho(\lambda \sigma.\mathcal{M}[\text{while } p s]_\rho k \sigma) \\
\mathcal{G}_J &: \mathcal{J}[\text{goto } l] = \lambda pk.() \\
\mathcal{L}_J &: \mathcal{J}[l : s] = \lambda pk.\text{mkenv}(\mathcal{J}[s]_\rho k, [l \mapsto (\lambda \sigma.\mathcal{M}[s]_\rho k \sigma)])
\end{align*}
\]

where:
\[
\text{send}(k, \sigma) = k \sigma
\]

**Figure 6.1** Continuation-Passing Semantics for goto
6.1.2 Analysis

We were unable to design a reasonable continuation semantics for goto that supported the staging transformations resulting in the pdt. The problem with this approach arose in the continuation argument. Specifically, the continuation encoded the control dependence relation in such a way that the extraction of the information was not possible using the staging transformation. While the extended direct semantics used in Chapter 5 is useful in representing general control operations [22], several problems surfaced with this approach in moving to the demand perspective. First, with general control flow, statements can be statically unreachable from the start of the program. Traditionally, these statements are considered dead-code, and most programmers and compiler writers expect that these statements are never executed. Since these statements may be reachable from the end of the program, however, demand driven processing from the end of the program cannot recognize this code as statically unreachable with the local information available. More importantly, the introduction of branching into and out of loops complicates the demand driven processing for loops. Thus, these approaches were abandoned.

6.2 Text Expansion Semantics for goto Statements

The basic approach for handling goto statements is a generalization of the expansion of the while loop performed for language W. The text at the target label of a goto statement replaces the goto statement itself. The unrolling of the goto statements, however, occurs on the syntactic level, producing programs with a potentially infinite number of statements. These infinite programs only contain if statements and assignment statements joined using the statement composition operator, with some annotations on the if statements. The statement annotations are described below.

The text expansion function appears in Figure 6.2. The expansion function resembles the continuation-passing semantics shown in Figure 6.1, except that the output of the function is a program. The label environment still maps statement labels to continuations, but continuations are now potentially infinite pieces of program text. The expansion annotates each if statement with the set of identifiers that are assigned in that statement. The help function A_f, defined in Figure 6.3, determines the finite set of identifiers that are assigned in a statement. Since expanded programs are potentially infinite pieces of text, the identifiers that are assigned within an if statement
must be specified separately. The $d$ argument in the text expansion function carries the assignment information for the statements in the continuation.

The assignment clause, $A_T$, composes the text in continuation $k$ with the text of the assignment statement. No composition is required if the continuation is empty, designated with the marker $\epsilon$. The if clause, $I_T$, builds an if statement annotated with the assignments for the two branches of the if statement. The identifiers assigned in the branches, combined with the set of identifiers assigned in the continuation, determine the sets with which to annotate the if statement. The if clause places statements in the continuation $k$ on both branches of the if statement. A while statement maps to an infinite series of if statements, each with the appropriate annotation for the assignments appearing in the body of the while statement and the continuation. The series of if statements forms a tree. The predicate representing the test for entering the loop is the root of the tree. The true branch of this if statement contains the body of the loop and the next if statement. For each of the if statements generated for the while statement, the false branch is the continuation $k$. The goto clause, $G_T$, discards the text in the continuation and uses instead the text at the target label of the goto, retrieved from the label environment, $\rho$. The clause for statement composition, $C_T$ builds a continuation from the text for the second statement in the composition.

The function $J$ creates the label environment used by $\mathcal{T}$. This label function is structurally the same as the function presented in Figure 6.1. However, it carries the assignment information with the continuation to invoke the expansion function, $\mathcal{T}$, and the store argument is not required.

The domain $stmt_d$ is a domain of potentially infinite syntax [53] containing infinite-text programs in addition to finite-text programs. The output of the $\mathcal{T}$ function is a potentially infinite program in language $\mathcal{W}$ without while statements and with each if statement annotated with the finite set of identifiers assigned on each branch of the statement. Using the annotations to determine where identifiers are assigned, the semantics provided in Chapter 3 for programs in language $\mathcal{W}$ applies to the infinite programs generated by the text expansion function.

The infinite text in these programs arise from loops in the program, either from while statements or from loops constructed explicitly with goto statements. With the exception of a program written with an explicit infinite loop of the form

\begin{verbatim}
1:  goto1
\end{verbatim}
Semantic Domains:

\[ P \in \text{pgm} \]
\[ Q \in \text{pgm}_a = \text{stmt}_a \otimes \text{end} \]
\[ s, s_i \in \text{stmt} \]
\[ c \in \text{stmt}_a = (\text{ide} \otimes \text{exp}) \oplus (\text{stmt}_a \otimes \text{stmt}_a) \]
\[ = (\text{def} \otimes \text{def} \otimes \text{exp} \otimes \text{stmt}_a \otimes \text{stmt}_a) \]
\[ x \in \text{ide} \]
\[ e, p \in \text{exp} \]
\[ l \in \text{lab} \]
\[ \rho \in \text{labenv} = \text{lab} \to \text{stmt}_a \]
\[ k \in \text{cont} = \text{stmt}_a \otimes e \]
\[ d \in \text{def} = \text{pf}_a(\text{ide}) \]

Semantics:

\[ T_p : \text{pgm} \to \text{pgm}_a \]
\[ J : \text{stmt} \to \text{labenv} \to \text{cont} \to \text{def} \to \text{labenv} \]
\[ T : \text{stmt} \to \text{labenv} \to \text{cont} \to \text{def} \to \text{stmt}_a \]

\[ T_p[s; \text{end}] = T[s] \rho e\emptyset; \text{end} \]
where \( \rho = \text{fix}(\lambda r.J[r]r\emptyset) \)

\[ \begin{align*}
A_T : \quad T[x := e] &= \lambda pkd.k \epsilon - x := e, x := e; k \\
I_T : \quad T[	ext{if } p \ s_t \ s_f] &= \lambda pkd.\text{if}(d \cup A_f(s_t, \rho), d \cup A_f(s_f, \rho)) \\
& \quad \quad \quad \quad p (T[s_t]pkd)(T[s_f]pkd) \\
C_T : \quad T[s_1; s_2] &= \lambda pkd.T[s_1]\rho(T[s_2]pkd)(d \cup A_f(s_2, \rho)) \\
W_T : \quad T[	ext{while } p \ s] &= \lambda pkd.\text{fix}(\lambda f.\text{if}(d \cup A_f(s, \rho), d) p (T[s]pkd) k) \\
G_T : \quad T[	ext{goto } l] &= \lambda pkd.\text{pl} \\
L_T : \quad T[l : s] &= \lambda pkd.T[s]pkd \\
A_J : \quad J[x := e] &= \lambda pkd.(e) \\
I_J : \quad J[	ext{if } p \ s_t \ s_f] &= \lambda pkd.\text{mkenv}(J[s_t]pkd, J[s_f]pkd) \\
C_J : \quad J[s_1; s_2] &= \lambda pkd.\text{mkenv}(J[s_1]\rho(T[s_2]pkd)(d \cup A_f(s_2, \rho)), J[s_2]pkd) \\
W_J : \quad J[	ext{while } p \ s] &= \lambda pkd.\text{J}[s]\rho(T[	ext{while } p \ s]pkd)(d \cup A_f(s, \rho)) \\
G_J : \quad J[	ext{goto } l] &= \lambda pkd.(e) \\
L_J : \quad J[l : s] &= \lambda pkd.\text{mkenv}(J[s]pkd, [l \mapsto T[s]pkd])
\end{align*} \]

**Figure 6.2** Text Expansion Function
the infinite portions of an expanded program appear as an infinite nesting of if statements. This structure is obvious in the case of the while loop expansion. In the goto case, if there is an exit from the loop, there must be an if statement in the body of statements forming the loop that determines whether the goto is evaluated. This if statement has the expansion of goto on one branch, creating the desired nesting structure. This structure of the expanded programs allows the demand semantics to proceed back through the infinite program.

6.2.1 Dependence Under Text Expansion

The text expansion function, \(T\), while yielding a program in the annotated language \(W\), introduces control dependences that do not correspond to dependences under the traditional notion of control dependence. The following program, \(M\), illustrates the problem:

\[
M = \text{if } p \\
    \text{goto 1} \\
    \quad \text{goto 2;}
\]

\[
1 : x := 1; \\
2 : y := 3; \\
\text{end}
\]

---

\[
A_f : stmt \rightarrow labenv \rightarrow P_{\text{fin}}(ide)
\]

\[
A_f[x := e] = \lambda \rho . \{x\}
\]

\[
A_f[\text{if } e \ s_l \ s_f] = \lambda \rho . A_f[s_l]_\rho \cup A_f[s_f]_\rho
\]

\[
A_f[s_1 ; s_2] = \lambda \rho . A_f[s_1]_\rho \cup A_f[s_2]_\rho
\]

\[
A_f[\text{while } e \ s] = \lambda \rho . A_f[s]_\rho
\]

\[
A_f[\text{goto}] = \lambda \rho . A_f[\rho]_\rho
\]

\[
A_f[\text{end}] = \lambda \rho . \emptyset
\]

**Figure 6.3** Definition of the \(A_f\) Function
This program expands to the following program using the text expansion function:

$$T[M] = \begin{cases} \text{if } p \\ x := 1; y := 3 \\ y := 3; \end{cases}$$

The $pdt$ for program $T[M]$ with $y$ as the demanded identifier contains two nodes for the assignment $y := 3$. However, these nodes have different control dependences: one has a true control dependence on $p$, the other a false control dependence on $p$. Under traditional definitions of control dependence, this assignment should be control dependent only on the start node since the assignment post-dominates the predicate node. The text expansion function disrupts the post-dominator relation, by lifting the assignment to $y$ into both branches of the if statement and thus alters the control dependence relation of the program.

However, in a sequential interpretation of $M$, if the predicate $p$ diverges, then the entire program denotes $\bot$. So there is a dependence in some sense of the assignment statement on the predicate $p$. The traditional control dependence semantics dominates the text expansion semantics of Figure 6.2, which we refer to as the lackadaisical semantics. The traditional relation is lazier, and thus less restrictive, than the lackadaisical semantics and captures, for control dependence, the demand driven character that underlies our characterization of data dependence. A control dependence should only exist if the predicate is required to determine the outcome of the program. If the node evaluates regardless of the outcome of the predicate, then the predicate is not required to determine under what conditions the statement should be evaluated. The post-dominator relation characterizes this situation. A statement $s_2$ post-dominates another statement $s_1$ iff $s_2$ is on all execution paths from statement $s_1$. Thus, a control dependence should not exist to a statement that post-dominates the predicate that is the source of the dependence. Just as non-terminating computations that do not affect the program demand are ignored in the lazy semantics for language $\mathcal{W}$, a diverging predicate should be ignored in a lazy semantics for goto if that predicate does not otherwise affect the final demand. In the above program $M$, the value of $y$ does not depend on which branch of the if statement applies; therefore, there should be no control dependence. The next section presents a semantics that captures the traditional notion of control dependence in the presence of goto statements.
6.3 Lazy Semantics for Goto Statements

Since the definition of control dependence in the presence of goto statements relies on the post-dominator relation, a change to the post-dominator relation clearly impacts the control dependence relation. Unfortunately, post-dominance is not a local property, even in an unexpanded program. Therefore, the denotational semantics in Figure 6.1 does not have enough information to determine the appropriate control dependences. To remedy this, we provide the semantics with the post-dominator relation for the original, unexpanded program, using statement numbers to correlate statements in the expanded programs with the statements in the unexpanded program. By specifying the text expansion function so that it retains these statement numbers for each instance of a statement in the expanded program, the post-dominator relation for the original program provides the proper information for statements in the expanded program. The modified text expansion function appears in Figure 6.4. This text expansion differs from the function presented in Figure 6.2 in the inclusion of the statement numbers. For assignment and if statements, the statement numbers are carried over into the expanded text. For a while loop, each instance of a statement that is generated carries the statement number of the statement in the unexpanded program. Statement numbers for goto statement are discarded; the numbers on the target statements are used instead. The assignment information for the if statements is augmented with the set of identifiers that are assigned values in assignment statements that do not post-dominate the predicate. This information is needed to properly place valve nodes.

6.3.1 Derivation of the Post-Dominator Relation

Statement \( b \) post-dominates statement \( a \) if \( b \) appears on all paths in the program from statement \( a \) to the end of the program. In the denotational semantics from Figure 6.1, the paths in a program from a particular statement are implicitly contained in the continuation argument. If a statement is on all control paths from another statement, it must be on both branches of any if statement in the continuation. This insight and the continuation-passing semantics are the basis for the function determining the post-dominator relation. Figure 6.5 contains a continuation-passing semantics for the language with goto statements. This semantics differs from the semantics in Figure 6.1 in two ways. First, statement numbers are assumed to be present on all statements in the source program, although statement numbers for statement
Semantic Domains:

\[ P \in \text{pgm} \]
\[ Q \in \text{pgm}_a = \text{stmt}_a \otimes \text{end} \]
\[ s, s_i \in \text{stmt} \]
\[ c \in \text{stmt}_a = (\text{snun} \otimes \text{ide} \otimes \text{exp}) \oplus (\text{stmt}_a \otimes \text{stmt}_a) \]
\[ x \in \text{ide} \]
\[ e, p \in \text{exp} \]
\[ l \in \text{lab} \]
\[ u \in \text{snun} \]
\[ \rho \in \text{labenv} = \text{lab} \rightarrow \text{stmt}_a \]
\[ k \in \text{cont} = \text{stmt}_a \oplus \epsilon \]
\[ d \in \text{def} = P_{\alpha}(\text{ide}) \]

Semantics:

\[ T_p : \text{pgm} \rightarrow \text{pgm}_a \]
\[ J : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{def} \rightarrow \text{labenv} \]
\[ T : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{def} \rightarrow \text{stmt}_a \]

\[ T_{p}[c; (u)\text{end}] = T[c]\rho \epsilon \emptyset; (u)\text{end} \]
where \( \rho = \text{fix}(\lambda r. J[c]r \epsilon \emptyset) \)

\[ A_T : T[(u)x := e] = \lambda pkd.k \downarrow \epsilon \rightarrow \emptyset; [(u)x := e].][(u)x := e]; k \]
\[ I_T : T[(u)\text{if } p s_t s_f] = \lambda pkd.(u)\text{if}(d \cup A_f(s_t), d \cup A_f(s_f), A_f(s_t) \cup A_f(s_f)) \]
\[ p (T[s_t\rho pkd](T[s_f\rho pkd]d)) \]
\[ C_T : T[s_1; s_2] = \lambda pkd.T[s_1\rho](T[s_2\rho pkd](d \cup A_f(s_2))) \]
\[ W_T : T[(u)\text{while } p s] = \lambda pkd.\text{fix}(\lambda f.[(u)\text{if}(d \cup A_f(s), d, A_f(s)) p](T[s\rho pkd]d) k) \]
\[ G_T : T[(u)\text{goto } l] = \lambda pkd.pl \]
\[ L_T : T[l : s] = \lambda pkd.T[s\rho pkd] \]

\[ A_J : J[(u)x := e] = \lambda pkd.() \]
\[ I_J : J[(u)\text{if } p s_t s_f] = \lambda pkd.\text{mkenv}(J[s_t\rho pkd], J[s_f\rho pkd]) \]
\[ C_J : J[s_1; s_2] = \lambda pkd.\text{mkenv}(J[s_1\rho](T[s_2\rho pkd](d \cup A_f(s_2))), J[s_2\rho pkd]) \]
\[ W_J : J[(u)\text{while } p s] = \lambda pkd.J[s]\rho(T[(u)\text{while } p s\rho pkd](d \cup A_f(s))) \]
\[ G_J : J[(u)\text{goto } l] = \lambda pkd.() \]
\[ L_J : J[l : s] = \lambda pkd.\text{mkenv}(J[s\rho pkd], [l \rightarrow T[s\rho pkd])] \]

**Figure 6.4** Text Expansion Function with Statement Numbers
compositions are ignored. These statement numbers are distinct from statement labels, and each statement has a unique statement number. Second, the clauses for the if and the while statements use a help function mkcont to form the required continuation.

The modified semantic function serves as the basis for a function to find the post-dominators using abstract interpretation. Figure 6.6 contains the result of these changes.

The desired type for the final output is

\[ \text{pdfun} = \text{sn} \mapsto P_f(s\text{num}), \]

a function mapping a statement number \( u \) to the set of statement numbers that post-dominate \( u \). The set of post-dominators is finite since the set of statements in the program is finite. The post-dominator function is finite for the same reason. Since the post-dominator relation is defined in terms of all the paths in the program, it is a static property and the abstract interpretation does not require the store to compute the relation.

The differences between the modified semantic function and the post-dominator function are predominantly in the continuation argument. Conceptually, the continuation argument encodes the result for the rest of the program. Thus, the continuation argument at any point in the semantics must contain a function characterizing the post-dominator relation for the remaining statements in the program. However, the result at any step must also have the information specifying which statements post-dominate the current statement. Specifically, the entry for the current statement depends on which statements in the continuation are on all paths from the current statement. Thus, there are two pieces of information in the continuation: the post-dominator information for all statements in the rest of the program and the statements in the rest of the program that appear on all paths from the current statement. The continuation in this meaning function is a pair of the following type:

\[ \text{cont} = P_f(s\text{num}) \otimes (s\text{num} \mapsto P_f(s\text{num})) \]

The first component is the set of statement numbers that are on all paths through the rest of the program; the second component contains the post-dominator function for the rest of the program.

The help function \( \text{send} \),

\[ \text{send}((u_\rho, \rho), u) = (u \cup u_\rho, \{(u, u \cup u_\rho)\} \cup \rho - \{(u, v_j) | (u, v_j) \in \rho\}) \]
Semantic Domains:

\[
P \in \text{pgm}
\]
\[
s, s_i \in \text{stmt}
\]
\[
x \in \text{ide}
\]
\[
e, p \in \text{exp}
\]
\[
l \in \text{lab}
\]
\[
u \in \text{nnum}
\]
\[
\rho \in \text{labenv} = \text{lab} \rightarrow \text{cont}
\]
\[
\sigma \in \text{store} = \text{ide} \rightarrow \text{val}
\]
\[
k \in \text{cont} = \text{store} \rightarrow \text{store}
\]

Semantics:

\[
M_p' : \text{pgm} \rightarrow \text{cont}
\]
\[
J : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{labenv}
\]
\[
M' : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{cont}
\]
\[
M_p'[s; (u)\text{end}] = \lambda \sigma. M'[s] \rho(\lambda \sigma. \sigma) \sigma
\]
where \(\rho = \text{fix}(\lambda r. J[s] r(\lambda \sigma. \sigma))\)

\[
A_M : M'[u]x := e \quad = \lambda pk. \sigma. \text{send}(k, \delta(\sigma, x, E[e]) \sigma)
\]
\[
I_M : M'[u]\text{if } p \text{ si } s_f \quad = \lambda pk. \sigma. \text{mkcont}(E[p] \sigma, M'[s_i] \rho \sigma, M'[s_f] \rho \sigma)
\]
\[
C_M : M'[s_1; s_2] \quad = \lambda pk. \sigma. M'[s_1] \rho(\lambda \sigma'. M'[s_2] \rho \sigma') \sigma
\]
\[
W_M : M'[u]\text{while } p \text{ s} \quad = \lambda pk. \text{fix}(\lambda f. \lambda \sigma'. \text{mkcont}(E[p] \sigma', M'[s] \rho f \sigma', \text{send}(k, \sigma')))
\]
\[
G_M : M'[u]\text{goto } l \quad = \lambda pk. \sigma. \text{send}(pl, \sigma)
\]
\[
L_M : M'[l] \quad = \lambda pk. \sigma. M'[s] \rho \sigma
\]
\[
A_J : J[u]x := e \quad = \lambda pk.() \quad \text{(same as } A_M)\]
\[
I_J : J[u]\text{if } p \text{ si } s_f \quad = \lambda pk. \text{mkenv}(J[s_1] \rho k, J[s_f] \rho k)
\]
\[
C_J : J[s_1; s_2] \quad = \lambda pk. \text{mkenv}(J[s_1] \rho (\lambda \sigma. M'[s_2] \rho k), J[s_2] \rho k)
\]
\[
W_J : J[u]\text{while } p \text{ s} \quad = \lambda pk. J[s] \rho(\lambda \sigma. M'[\text{while } p \text{ s}] \rho k \sigma)
\]
\[
G_J : J[u]\text{goto } l \quad = \lambda pk.() \quad \text{(same as } G_M)\]
\[
L_J : J[l] \quad = \lambda pk. \text{mkenv}(J[s] \rho k, [l \mapsto (\lambda \sigma. M'[s] \rho k \sigma)])
\]

where:

\[
\text{send}(k, \sigma) \quad = \quad k \sigma
\]
\[
\text{mkcont}(v, \sigma_i, \sigma_f) \quad = \quad v \rightarrow \sigma_i, \sigma_f
\]

**Figure 6.5** Modified Continuation-Passing Semantics
adds a new entry for the statement number \( u \) to the post-dominator function \( \rho \) using
the information in the first component of the continuation. In addition, the statement
number \( u \) is added to the set of statements on all paths from this statement.

The help function \( mkcont \),

\[
\text{mkcont}(u, (u_t, \rho_t), (u_f, \rho_f)) = (u \cup (u_t \cap u_f), \{ (u, u \cup (u_t \cap u_f)) \} \cup \rho_t \cup \rho_f - \\
\{(u, v_j) \mid (u, v_j) \in \rho_t \cup \rho_f \})
\]

combines the continuation for the true and the false branches to make a new contin-
uation. The intersection of the set of statement numbers on all paths from the true
branch, \( u_t \), and the set for the false branch, \( u_f \), unioned with the statement number
of the predicate \( u \) is the set of statement numbers reachable on all paths from the
predicate statement.

The clauses in the post-dominator function reflect the change in the type of the
send and \( mkcont \) functions as well as the removal of the store argument. No other
changes to the continuation-passing semantics are necessary to give a function that
determines the post-dominator relation for a program.

**Theorem 6.1** Let \( O = \tilde{M}_p[P] \) for program \( P \). The statement num-
bered \( u \) in program \( P \) post-dominates the statement numbered \( u' \) iff
\( u \in O(u') \).

**Proof** The function \( \tilde{M} \) traces all paths in a program that begin at the
start of the program. Paths split only at if and while statements, the
points of conditional control flow. \( \tilde{M} \) intersects the result from these two
paths to find the statements common to both paths.

For the only if direction, assume that \( u \in O(u') \) and \( u \) does not post-
dominate \( u' \). Since \( u \) does not post-dominate \( u' \), there is some path around
\( u \) from \( u' \). Thus there exists a predicate statement on that path such that
\( u \) is not on one of the paths of that predicate. However, if \( u \in O(u') \) then
it was on both paths from that predicate, a contradiction.

For the if direction, assume that \( u \) post-dominates \( u' \). By the definition
of the post-dominator relation, \( u \) appears on all paths from \( u' \) in the
evaluation of the program. Therefore, it is on all paths from any predicate
encountered and is included in the set of nodes reachable from \( u' \) since it
is in both of the input sets of the intersection function. \( \square \)
Semantic Domains:

\[ P \in \text{pgm} \]
\[ s, s_i \in \text{stmt} \]
\[ x \in \text{ide} \]
\[ e, p \in \text{exp} \]
\[ l \in \text{lab} \]
\[ u \in \text{snun} \]
\[ \mathcal{O} \in \text{pdfun} = \text{snun} \rightarrow P_{\text{fn}}(\text{snun}) \]
\[ \rho \in \text{labenv} = \text{lab} \rightarrow \text{cont} \]
\[ k \in \text{cont} = P_{\text{fn}}(\text{snun}) \odot \text{pdfun} \]

Semantics:

\[ \tilde{M}_p : \text{pgm} \rightarrow \text{pdfun} \]
\[ \mathcal{J} : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{labenv} \]
\[ \tilde{M} : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{cont} \]

\[ \tilde{M}_p[s; (u)\text{end}] = \text{right}(\tilde{M}[s]\rho\emptyset) \]
where \[ \rho = \text{fix}(\lambda\rho'.\mathcal{J}[s]\rho') \]

\[ \text{A}_M : \tilde{M}[(u)x := e] = \lambda pk.\text{send}(k, u) \]
\[ \text{I}_M : \tilde{M}[(u)\text{if } p s_i s_f] = \lambda pk.\text{mkcont}(u, \tilde{M}[s_i]\rho k, \tilde{M}[s_f]\rho k) \]
\[ \text{C}_M : \tilde{M}[s_1; s_2] = \lambda pk.\tilde{M}[s_1]\rho k(\tilde{M}[s_2]\rho k) \]
\[ \text{W}_M : \tilde{M}[(u)\text{while } p s] = \lambda pk.\text{fix}(\lambda f.\text{mkcont}(u, \tilde{M}[s]\rho f, \text{send}(k, u))) \]
\[ \text{G}_M : \tilde{M}[(u)\text{goto } l] = \lambda pk.\text{send}(pl, u) \]
\[ \text{L}_M : \tilde{M}[l : s] = \lambda pk.\tilde{M}[s]\rho k \]

\[ \text{A}_J : \mathcal{J}[(u)x := e] = \lambda pk.(()) \]
\[ \text{I}_J : \mathcal{J}[(u)\text{if } p s_i s_f] = \lambda pk.\text{mkenv}(\mathcal{J}[s_i]\rho k, \mathcal{J}[s_f]\rho k) \]
\[ \text{C}_J : \mathcal{J}[s_1; s_2] = \lambda pk.\text{mkenv}(\mathcal{J}[s_1]\rho k(\mathcal{M}[s_2]\rho k), \mathcal{J}[s_2]\rho k) \]
\[ \text{W}_J : \mathcal{J}[(u)\text{while } p s] = \lambda pk.\mathcal{J}[s]\rho k(\mathcal{M}[\text{while } p s]\rho k) \]
\[ \text{G}_J : \mathcal{J}[(u)\text{goto } l] = \lambda pk.(()) \]
\[ \text{L}_J : \mathcal{J}[l : s] = \lambda pk.\text{mkenv}(\mathcal{J}[s]\rho k, [l \mapsto \tilde{M}[s]\rho k]) \]

send((u_\rho, \rho), u) = (u \cup u_\rho, \{(u, u \cup u_\rho)\} \cup \rho - \{(u, v_j) \mid (u, v_j) \in \rho\})

mkcont(u, (u_t, \rho_t), (u_f, \rho_f)) = (u \cup (u_t \cup u_f), \{(u, u \cup (u_t \cup u_f))\} \\
\cup \rho_t \cup \rho_f - \{(u, v_j) \mid (u, v_j) \in \rho_t \cup \rho_f\})

---

**Figure 6.6** Post-Dominator Functions
6.3.2 Control Dependence

There are two requirements for statement $a$ to be control dependent on statement $b$. First, statement $a$ must follow statement $b$ in some evaluation of the program. Second, $a$ must not strictly post-dominate $b$.\textsuperscript{13} The demand semantics traces, for expanded programs, essentially the same paths as those traversed by program execution. Specifically, the denotation of statement $a$ will not rely in any way on the denotation of $b$ unless statement $a$ follows statement $b$ on some evaluation path of the program. Thus, any statement that is examined in the denotation of $a$ directly satisfies the first criteria for control dependence. Any control statement $b$ examined is thus a potential control dependence predecessor for $a$. If statement $a$ strictly post-dominates statement $b$, $b$ is not a control dependence predecessor for $a$, and this statement can be ignored in determining the denotation of $a$. Using this fact, the control stage for language $\mathcal{W}$ is modified to check the post-dominator relation before using the predicate information to determine if the statement is enabled for evaluation. Figure 6.7 contains the new semantic functions for the control stage of the derivation. The data dependence stage is unchanged.

This semantic function does not include clauses for \texttt{while} or \texttt{goto} statements since the text expansion function replaces these statements with the expansion of the loop and the program text associated with the target of the \texttt{goto} respectively. The composition and \texttt{end} clauses are unchanged except for the addition of the post-dominator argument, $\mathcal{O}$. The assignment clause, $A_C$, uses the auxiliary function $ck$ to determine which of the predicate values in the control parameter affect the evaluation of the assignment statement and what those values are. The rest of the clause is unchanged.

In the control stage for language $\mathcal{W}$, control parameters are boolean values. These values are combined using the logical \texttt{and} operation, as the control structures are traversed. In the presence of non-structured control flow, however, some boolean values must be bypassed to reflect the lazy control dependence relation. Thus, control parameters must contain not only the boolean value but also the source of the value. Control parameters in the modified semantic functions are sequences of pairs of statement numbers and boolean values. The source of the statement number is the \texttt{if} or \texttt{while} statement in the original program that contained the predicate expression whose value is the boolean value. The \texttt{if} clause, $I_C$ builds these control parameter

\textsuperscript{13}This is a slightly modified definition of control dependence, but is shown equivalent in [20].
Semantic Domains:

\[ v \in \text{val}^T = \text{val} \cup \{\top\} \]
\[ b \in \text{bool}^T = \text{bool} \cup \{\top\} \]
\[ i, x \in \text{ide} \]
\[ u \in \text{sn} \]
\[ O \in \text{pdfun} = \text{sn} \rightarrow P_{fn}(\text{sn}) \]
\[ e, p \in \text{exp} \quad \text{(abstract)} \]
\[ \kappa \in \text{eval} = \text{true}_c \oplus ((\text{sn} \otimes \text{bool}^T) \otimes \text{eval}) \]
\[ \sigma \in \text{store}^T = \text{ide} \rightarrow \text{val}^T \]

Semantics:

\[ \mathcal{E} : \text{exp} \rightarrow \text{store} \rightarrow \text{val}^T \quad \text{(abstract)} \]
\[ \mathcal{C} : \text{stmt}_o \rightarrow \text{pdfun} \rightarrow \text{ide} \rightarrow \text{eval} \rightarrow \text{store}^T \rightarrow \text{val}^T \]

\[ \mathcal{A}_C : \mathcal{C}[(u)\mathcal{E}x := e] = \lambda \text{Oik} \cdot \text{.} \]
\[ \mathcal{I}_C : \mathcal{C}[(u)\mathcal{I}(d_i, d_f, d_o) p \mathcal{S} i \mathcal{S} f] = \lambda \text{Oik} \cdot \text{.} \]
\[ \text{let } \kappa^+ = ((u, \mathcal{E}[p]\sigma), \kappa), \]
\[ \kappa^- = ((u, -\mathcal{E}[p]\sigma), \kappa) \text{ in} \]
\[ (i \notin d_i \rightarrow (\mathcal{V}(T, p, \sigma, O, u, \kappa) \rightarrow \sigma_i, \bot)), \]
\[ \mathcal{C}[s_i] \text{Oik}^+(\lambda j. j \notin d_o \rightarrow \sigma_j, \]
\[ \mathcal{V}(T, p, \sigma, O, u, \kappa) \rightarrow \sigma_j, \bot)) \]
\[ \cup \]
\[ (i \notin d_f \rightarrow (\mathcal{V}(F, p, \sigma, O, u, \kappa) \rightarrow \sigma_i, \bot)), \]
\[ \mathcal{C}[s_j] \text{Oik}^-(\lambda j. j \notin d_o \rightarrow \sigma_j, \]
\[ \mathcal{V}(F, p, \sigma, O, u, \kappa) \rightarrow \sigma_j, \bot)) \]

\[ \mathcal{C}_C : \mathcal{C}[s_1, s_2] = \lambda \text{Oik} \cdot \text{.} \]
\[ \mathcal{F}_C : \mathcal{C}[s; \mathcal{E}(u)\text{end}] = \lambda \text{Oik} \cdot \text{.} \]

where

\[ \mathcal{C}(O, u, \text{true}_c) = \top \]
\[ \mathcal{C}(O, u, [(\bar{u}, b), \kappa']) = u \in O(\bar{u}) \land u \neq \bar{u} \rightarrow \mathcal{C}(O, u, \kappa'), b \land \mathcal{C}(O, \bar{u}, \kappa') \]

\[ \mathcal{V}(T, p, \sigma, O, u, \kappa) = \mathcal{E}[p]\sigma \land \mathcal{C}(O, u, \kappa) \]
\[ \mathcal{V}(F, p, \sigma, O, u, \kappa) = -\mathcal{E}[p]\sigma \land \mathcal{C}(O, u, \kappa) \]

**Figure 6.7** Control Stage for Language \( \mathcal{W} \)
sequences. The help function $ck$ determines which pairs in a control parameter sequence apply to the statement under consideration. The function $ck(O, u, [(\bar{u}, b), \kappa'])$ first determines if statement $u$ strictly post-dominates $\bar{u}$. If it does, $u$ is not control dependent on the predicate statement $\bar{u}$, the predicate value $b$ is ignored, and the rest of the control node is checked. If $u$ does not strictly post-dominate $\bar{u}$, the value $b$ is combined with the result for the control predecessors of the predicate statement $\bar{u}$, $\kappa'$, to determine if $u$ is enabled for evaluation.

The if clause, $I_c$, differs from the clause for language $\mathcal{W}$ in the construction of the control nodes and in the control parameters that are required for the valve statements. The help function $vp$ builds the control information for the valve statements. The control predicate built for the true branch, $\kappa^+$, is the pair $(u, \mathcal{E}[p]\sigma)$ consed onto the input control parameter $\kappa$. The control parameter $\kappa^-$ is constructed similarly. If the demanded identifier is not assigned on the true branch of the if statement, the value from the store is the result when the predicate evaluates to true and the if statement itself is enabled. These conditions are verified using the value $ck(O, u, \kappa)$. By construction, the valve statement is control dependent on the if statement, and requires the value of the predicate expression $p$. The store for each of the branches also contains the valve predicate information. This is necessary since the removal of the predicate information for statements post-dominating the predicate allows demands to be propagated into an if branch through this store which does not assign a value for the demanded identifier. If the demand is not satisfied within that branch, the modified store creates the valve statement to properly gate the value from before the if statement.

As in the case of the other languages, no program denotes $\top$.

**Theorem 6.2** For program $P$, identifier $i$, boolean value $\kappa$ and store $\sigma$ such that $\forall j. \sigma j \neq \top$ and with $O = \bar{M}_p[P]$,

$$C[P]\sigma i \kappa \neq \top$$

**Proof** We must use a strengthened induction hypothesis to prove the limit case:

Let $\{p_k \mid k \geq 0\}$ be a chain of programs such that $p_k \subseteq p_{k+1}$ for $k \geq 0$ and $P = \bigcup\{p_k\}$. Then, for all $k$

1. $C[p_k]\sigma i \kappa \neq \top$
2. $C[p_k]O_i\kappa \sigma = \bot \lor (C[p_k]O_i\kappa \sigma = v \land v \neq \bot \lor C[p_{k+1}]O_i\kappa \sigma = v)$

**Note:** This new induction hypothesis claims that, for each finite program in the ascending chain of programs approximating $P$, once the meaning of a $p_k$ is defined (denotes a non-bottom value), all of the programs above $p_k$ have the same meaning under this store. □_{Note}

Using this strengthen induction hypothesis, the limit case follows immediately since the lub of a set $\{\bot, v\}$ is $v$ so $\cup \{C[p_k]O_i\kappa \sigma\} = v$ and $v \neq \top$. We prove the induction hypothesis by induction on $k$ and on the structure of $p_k$.

**Case** $(u)x := e$ This case follows directly from the definitions of $E$ and $C$ and the assumption on the store $\sigma$.

**Case** $(u)\text{if}(d_t, d_f, d_o)\ p\ s_t\ s_f$ The result follows directly from the assumptions on the store $\sigma$ if $i \notin d_t \cup d_f$. Otherwise, there are several cases to consider.

Assume that $i$ is assigned only on one branch. Without loss of generality, assume that the assignment appears on the true branch. The assignment statement on the true branch does not post-dominate the if statement $u$ by the definition of post-dominance and the text expansion function. Thus, for statement $u_t$, $ck(O, u_t, [(u, p), \kappa])$ is false if the predicate $p$ denotes false. Thus, using Lemma 6.1, the result for the true branch is $\bot$ and the result holds. If the predicate $p$ denotes true, the result is immediate from the definition of $C$ and the induction hypothesis.

If $i$ is assigned on both branches, the situation is complex. The simple case holds if all the assignment statements for $i$ in $s_t$ and $s_f$ do not post-dominate the if statement $u$. In this case, the reasoning used above holds, at least one branch returns $\bot$, and the result is established.

Otherwise, it is possible that both branches denote a value. Let $u_t$ be the statement on the true branch and $u_f$ the statement on the false branch. By the definition of post-dominance, the demand propagation and the properties of the text expansion function and
since these statements both post-dominate the predicate, they must originate from the same statement and thus have the same expression component. For each free identifier in this expression, the following cases must be considered:

1. The statement satisfying the demands post-dominates the predicate. In this case, by the definition of post-dominance and the properties of the text expansion and demand propagation, the same statement satisfies the demand from \( u_i \) and \( u_f \). A proof by induction shows that the value for this identifier is the same for both demands.

2. The statement satisfying both the demands precedes the predicate. By the definition of \( C \), the value for at least one of the demands is \( \perp \) or the values are equal since they come from the same store.

3. The statements satisfying the demands do not post-dominate the predicate. In this case, at least one of the values will be \( \perp \) by Lemma 6.1.

Therefore, by the properties of expression evaluation, if both expressions do denote a value, the values are equal. Otherwise, one of the values is \( \perp \) and the result holds.

Now, we examine the implication in Part 2 of the theorem as it applies to the trees of if statements built for loops (the result is obvious for statement compositions). By the properties of the text expansion, the if statements represent the predicates for the loop iterations start with the zero iteration at the root of the tree and move to the first, second, etc. iterations as the tree is traversed from the root. By the construction of the predicates for each iteration, once the loop terminates (with a false predicate value), all subsequent iterations also have a false predicate value and that branch of the tree denotes \( \perp \). Thus, adding additional iterations to the tree past the point where the loop terminates only adds \( \perp \) values to the lub operations. The value found when the tree contains enough iterations to hold the terminating predicate never changes with the addition of these iterations.
Case $s_1; s_2$ Using the induction hypothesis, the store $\lambda j. C[[s_2]]j\kappa\sigma$ fulfills the assumptions of the theorem. Thus, the induction hypothesis holds for statement $s_2$ and the result is shown. 

Lemma 6.1 Let $s$ be a statement, $i$ an identifier, and $\sigma$ a store. Then,

1. If $A(s, i)$ is true and $ck(\mathcal{O}, u, \kappa) = false$ for all statements $u$ in $s$ that assign a value for $i$ then

$$C[[s]] i\kappa\sigma = \bot$$

2. If $A(s, i)$ is false, then

$$C[[s]] i\kappa\sigma = \sigma i$$

Proof The reasoning follows the same path as that for the proof of Lemma 3.1. The condition in the lemma is strengthened so that the dependence yielding the false value is not filtered out from consideration. The predicate information in the store is required in this proof. The limit step is trivial since all components of the set for which the lub is required are $\bot$. 

The following theorem characterizes the relationship between the semantic function $C$ and the function $\mathcal{V}$ for this language:

Theorem 6.3 For all programs $P$, stores $\sigma$ and $\bar{\sigma}$ such that:

$$\forall j. \bar{\sigma}j \subseteq \sigma j$$

and for all identifiers $i$,

$$\mathcal{V}[T_p\{P\}]i\bar{\sigma} \subseteq C[[T_p\{P\}]](M_p[[P]])i\text{true}_e\sigma$$

Proof The proof proceeds by fixed point induction on the structure of $P$ using the following induction hypothesis:
For a statement $s$, stores $\sigma$ and $\bar{\sigma}$ such that:

$$\forall j. \overline{\sigma}j \sqsubseteq \sigma j$$

and for identifier $i$, and for $\kappa$ such that $ck(O, fl(s), \kappa)$ is true where $fl(s)$ is the leftmost statement number in statement $s$, then

$$\forall[s]i\sigma \sqsubseteq C[s]Oi\kappa\sigma$$

where $O$ reflects the post-dominator relation for statement $s$.

The induction hypothesis is an admissible predicate[28], so fixed point induction is applicable.

**Case** $(u)x := e$ The applicable denotations are as follows:

$$\forall[(u)x := e]i\sigma = i \neq x \to \epsilon[e]i\sigma, \bar{\sigma}i$$

$$\forall[(u)x := e]Oi\kappa\sigma = i \neq x \to (ck(O, u, \kappa) \to \epsilon[e]\sigma, \bot), \sigma i$$

These denotations are the same given the assumption on the stores and the assumption that $ck(O, u, \kappa)$ is true.

**Case** $s_1; s_2$ The denotations for statement composition are as follows:

$$\forall[s_1; s_2]i\sigma = \forall[s_2]i(\lambda j. \forall[s_1]j\sigma)$$

$$\forall[s_1; s_2]Oi\kappa\sigma = \forall[s_2]Oi(\lambda j. C[s_1]oj\kappa\sigma)$$

It is only necessary to show that the stores in the two clauses for $s_2$ satisfy the assumptions in the induction hypothesis. By applying the induction hypothesis to the values $\forall[s_1]j\sigma$ and $C[s_1]Oj\kappa\sigma$, the stores can be shown to satisfy the assumptions. Using the induction hypothesis again, the theorem is established.

**Case** $(u)\text{if}(d_t, d_f, d_o) p s_t s_f$ The applicable denotations are as follows:

$$\forall[(u)\text{if}(d_t, d_f, d_o) p s_t s_f]i\sigma = \epsilon[p]i\sigma \to \forall[s_t]i\sigma, \forall[s_f]i\sigma$$
and

\[
C[(u)\textbf{if}(d_t, d_f, d_o) p s_t s_f] O i \kappa \sigma =
\]

\[
i \notin d_t \cup d_f \rightarrow \sigma i,
\]

let \( \kappa^+ = (u, \mathcal{E}[p] \sigma) \circ \kappa, \kappa^- = (u, -\mathcal{E}[p] \sigma) \circ \kappa \) in

\[
(i \notin d_t \rightarrow ((\mathcal{E}[p] \sigma \land ck(O, u, \kappa)) \rightarrow \sigma i, \bot),
\]

\[
C[s_t] O i \kappa^+(\lambda j. j \notin d_o \rightarrow \sigma j,
\]

\[
vp(T, p, \sigma, O, u, \kappa) \rightarrow \sigma j, \bot))
\]

\[
\cup
\]

\[
(i \notin d_f \rightarrow ((\mathcal{E}[p] \sigma \land ck(O, u, \kappa)) \rightarrow \sigma i, \bot),
\]

\[
C[s_f] O i \kappa^-(\lambda j. j \notin d_o \rightarrow \sigma j,
\]

\[
vp(F, p, \sigma, O, u, \kappa) \rightarrow \sigma j, \bot))
\]

There are three cases to consider. If the demanded identifier is not assigned on either branch, the result for \( C \) is \( \sigma i \). The result for \( V \) is \( \bot \) if the evaluation of the predicate \( p \) diverges or if the evaluation of the selected branch diverges, or \( \sigma i \) otherwise. The theorem holds regardless.

If the demanded identifier is assigned on both branches of the \textbf{if} statement, the result holds by the induction hypothesis and the fact that the assignment on at least one branch must denote \( \bot \) or the values returned will be equal, by the construction of the control parameters, the text expansion function and the strictness of the expression evaluation function. The reasoning here is the same as that used in Theorem 6.2.

If the demanded identifier is assigned on one branch only, the result holds using the induction hypothesis, the assumption on the store, and the fact that at least one branch must denote \( \bot \), as in the case above.

\( \square \)

6.3.3 Program Dependence Trees

The abstraction stage of the semantics proceeds as in the derivation for language \( W \). Figure 6.8 contains the semantic function \( C' \). The function \( ck' \) requires the initial store
to reduce the control parameters to boolean values. The control parameter sequences are pairs of statement numbers and functions from stores to boolean values. This stage of the transformation does not alter the denotations of programs, as shown in the following theorem:

**Theorem 6.4** For program $P$, stores $\sigma$ and $\bar{\sigma}$ and abstracted store $\gamma$ such that:

$$\forall j. \bar{\sigma}j \subseteq \gamma j \sigma$$

control sequences $\bar{\kappa}$ and $\kappa$ such that

$$ck(\mathcal{O}, u, \bar{\kappa}) = ck'(\mathcal{O}, u, \kappa, \sigma)$$

and for all identifier $i$,

$$C[\mathcal{P}](\mathcal{M}_p[\mathcal{P}])i\bar{\kappa} = C'[\mathcal{P}](\mathcal{M}_p[\mathcal{P}])i\kappa(\lambda j.\sigma j)\sigma$$

**Proof** The proof proceeds by fixed point induction on the structure of $P$ and is a straightforward extension of the proof for language $\mathcal{W}$. The induction hypothesis is an admissible predicate, so fixed point induction applies. $\square$

The final stage of the derivation creates the $pdt$ for a program. Figure 6.10 contains the function $\mathcal{G}$. The functions $\mathcal{P}$ and $\mathcal{P}_k$ are unchanged but appear in Figure 6.10 for ease of reference. Figure 6.9 gives the domain definitions for these functions. The clause for assignment uses the $ck$ function to select only those control nodes that are control dependence predecessors. The $\textbf{if}$ clause pairs the new control nodes with the statement number for the outermost predicate expression. The construction of the control node for valve nodes filters the control node of the $\textbf{if}$ statement itself. The other clauses simply reflect the addition of the post-dominator parameter.

The function $ck^+(\mathcal{O}, u, (\bar{u}, (t, \bar{\kappa}, p, c_p)))$ uses the post-dominator information $\mathcal{O}$ to determine if the predicate $p$ affects the denotation of statement $u$. If $u$ post-dominates $\bar{u}$, the predicate $p$ is not relevant and this part of the control node is ignored. Otherwise, this predicate information is retained and the control node for $p$ is processed to find the control predecessors for $\bar{u}$.

This stage of the derivation also retains the semantics of the program, as shown in the following theorem.
Semantic Domains:

\[ v \in \text{val}^T = \text{val} \cup \{ \top \} \quad \text{(abstract)} \]
\[ b \in \text{store} \rightarrow \text{bool}^T \]
\[ i, x \in \text{id} \]
\[ u \in \text{snun} \]
\[ O \in \text{pdfun} = \text{snun} \rightarrow P_{\beta_n}(\text{snun}) \]
\[ \kappa \in \text{cval} = \text{true}_c \oplus ((\text{snun} \otimes \text{store}^T \rightarrow \text{bool}^T) \otimes \text{cval}) \]
\[ e, p \in \text{exp} \quad \text{(abstract)} \]
\[ \sigma \in \text{store}^T = \text{ide} \rightarrow \text{val}^T \]

Semantics:

\[ \mathcal{E} : \text{exp} \rightarrow \text{store} \rightarrow \text{val}^T \quad \text{(abstract)} \]
\[ \mathcal{C}' : \text{stmt}_a \rightarrow \text{pdfun} \rightarrow \text{ide} \rightarrow (\text{store}^T \rightarrow \text{cval}) \rightarrow (\text{ide} \rightarrow \text{store}^T \rightarrow \text{val}^T) \]
\[ \quad \rightarrow \text{store}^T \rightarrow \text{val}^T \]

\[ A_{C'} : \mathcal{C}'[x := e] = \lambda Oi \kappa \gamma \lambda \sigma. \text{ck}'(O, u, \kappa, \sigma) \rightarrow \]
\[ (i \not= x \rightarrow \mathcal{E}[e](\lambda j. \gamma j \sigma), \gamma i \sigma), \bot) \]

\[ I_{C'} : \mathcal{C}'[\text{if}(d_i, d_j, d_o) \ p \ s_l \ s_f] = \lambda Oi \kappa \gamma \lambda \sigma. \text{ck}'(O, u, \kappa) \rightarrow \]
\[ \text{let } \kappa^+ = (u, \lambda \sigma. \mathcal{E}[p](\lambda j. \gamma j \sigma)) \circ \kappa, \]
\[ \kappa^- = (u, \lambda \sigma. \mathcal{E}[p](\lambda j. \gamma j \sigma)) \circ \kappa \text{ in} \]
\[ (i \not= d_i \rightarrow \]
\[ (vp'(T, p, \gamma, \sigma, O, u, \kappa) \rightarrow \gamma i \sigma, \bot), \]
\[ \mathcal{C}'[s_l] \mathcal{O}i \kappa^+ (\lambda j \sigma'. j \not= d_o \rightarrow \gamma j \sigma, \]
\[ vp'(T, p, \gamma, \sigma, O, u, \kappa) \sigma' \rightarrow \gamma j \sigma, \bot) \]
\[ \cup \]
\[ (i \not= d_j \rightarrow \]
\[ (vp'(F, p, \gamma, \sigma, O, u, \kappa) \rightarrow \gamma i \sigma, \bot), \]
\[ \mathcal{C}'[s_f] \mathcal{O}i \kappa^- (\lambda j \sigma'. j \not= d_o \rightarrow \gamma j \sigma, \]
\[ vp'(F, p, \gamma, \sigma, O, u, \kappa) \sigma' \rightarrow \gamma j \sigma, \bot) \]

\[ C_{C'} : \mathcal{C}'[s_1; s_2] = \lambda Oi \kappa \gamma. \mathcal{C}'[s_2] \mathcal{O}i \kappa (\lambda j. \mathcal{C}'[s_1] \mathcal{O}j \gamma) \]
\[ C_{C'} : \mathcal{C}'[s; (u)\text{end}] = \lambda Oi \kappa \gamma. \mathcal{C}'[s] \mathcal{O}i \kappa \gamma \]

where

\[ \text{ck}'(O, u, \text{true}_c, \sigma) = \text{T} \]
\[ \text{ck}'(O, u, [(\bar{u}, b), \kappa'], \sigma) = u \in O(\bar{u}) \land u \not= \bar{u} \rightarrow \text{ck}'(O, u, \kappa', \sigma), b \sigma \land \text{ck}'(O, \bar{u}, \kappa', \sigma) \]
\[ vp'(T, p, \gamma, \sigma, O, u, \kappa) = \lambda \sigma. \mathcal{E}[p](\lambda j. \gamma j \sigma) \land \text{ck}'(O, u, \kappa, \sigma) \]
\[ vp'(F, p, \gamma, \sigma, O, u, \kappa) = \lambda \sigma. \neg \mathcal{E}[p](\lambda j. \gamma j \sigma) \land \text{ck}'(O, u, \kappa, \sigma) \]

**Figure 6.8** Abstraction Step of the Staging Derivation
Figure 6.9  Domains for Program Dependence Trees

Theorem 6.5  For program $P$ and identifier $i$,

$$\mathcal{P}(\mathcal{G}[P](\tilde{M}_p[P])i(\text{true}_c)\gamma_0)\sigma_0 = \mathcal{C}'[P](\tilde{M}_p[P])i(\lambda_\sigma.T)(\lambda i_\sigma.i)\sigma_0$$

where

$$\gamma_0 = (\lambda i.(\text{true}_c, i))$$

Proof  The proof proceeds by fixed point induction on $P$ and requires the following induction hypothesis:

For a statement $s$, identifier $i$, store $\sigma$, abstracted store $\tilde{\gamma}$ and code table $\gamma$ such that:

$$\forall j. \tilde{\gamma}j\sigma = \mathcal{P}[\gamma j]\sigma$$

and for $\kappa$ and $\bar{\kappa}$ such that

$$ck'(O, fl(s), \bar{\kappa}, \sigma) = \mathcal{P}_k[ck'(O, fl(s), \kappa)]\sigma$$

where $fl(s)$ is the leftmost statement number in statement $s$, then

$$\mathcal{P}(\mathcal{G}[s]O\kappa\gamma)\sigma = \mathcal{C}'[s]O\kappa\tilde{\gamma}\sigma$$
Semantics:

\[
\begin{align*}
\mathcal{G} : \text{stmt} &\rightarrow \text{pdf} \rightarrow \text{ide} \rightarrow \text{p-node} \rightarrow \text{code-tbl} \rightarrow \text{code} \\
\mathcal{E} : \text{exp} &\rightarrow \text{store} \rightarrow \text{val}^T \\
\mathcal{P} : \text{code} &\rightarrow \text{store}^T \rightarrow \text{val}^T \\
\mathcal{P}_k : \text{p-node} &\rightarrow \text{store}^T \rightarrow \text{bool}^T \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{G}[u][x := e] &\quad = \lambda\sigma.\mathcal{G}[u](x = e) \\
\mathcal{G}[u][\text{if}(d, d_f, d_o) p s_t s_f] &\quad = \lambda\sigma.\mathcal{G}[u](d \not\in d_t \cup d_f \Rightarrow \gamma, \\
&\quad \quad \quad \text{let } \kappa^+ = (u, (T, \kappa, p, \mathcal{MS}(p, \gamma))), \\
&\quad \quad \quad \quad \kappa^- = (u, (F, \kappa, p, \mathcal{MS}(p, \gamma))) \text{ in} \\
&\quad \quad \quad (i \not\in d_t \Rightarrow \mathcal{MV}(p, \kappa, O, u, T, i, \gamma), \\
&\quad \quad \quad \mathcal{G}[s_t][\mathcal{O}i \kappa^+](\lambda_j. j \not\in d_o \Rightarrow \gamma_j, \mathcal{MV}(p, \kappa, O, u, T, j, \gamma)), \\
&\quad \quad \quad i \not\in d_f \Rightarrow \mathcal{MV}(p, \kappa, O, u, F, i, \gamma), \\
&\quad \quad \quad \mathcal{G}[s_f][\mathcal{O}i \kappa^-](\lambda_j. j \not\in d_o \Rightarrow \gamma_j, \mathcal{MV}(p, \kappa, O, u, F, j, \gamma))) \\
\mathcal{G}[s_1; s_2] &\quad = \lambda\sigma.\mathcal{G}[s_2]\mathcal{O}i\mathcal{K}(\lambda_j. \mathcal{G}[s_1]\mathcal{O}i\gamma) \\
\mathcal{G}[s; \text{end}] &\quad = \lambda\sigma.\mathcal{G}[s]\mathcal{O}i\gamma \\
\mathcal{P}[(\kappa, e, C_0)] &\quad = \lambda\sigma.\mathcal{P}_k[\kappa]\sigma \rightarrow \mathcal{E}[e]\mathcal{RS}(C_0, \sigma), \perp \\
\mathcal{P}[(\kappa, e, C_0, C_0)] &\quad = \lambda\sigma.\mathcal{P}_k[\kappa]\sigma \rightarrow (\mathcal{P}[C_0]\sigma \perp \perp \Rightarrow \perp, \mathcal{E}[e]\mathcal{RS}(C_0, \sigma)), \perp \\
\mathcal{P}[(C_t, C_f)] &\quad = \lambda\sigma.\mathcal{P}_k[C_t]\sigma \cup \mathcal{P}[C_f]\sigma \\
\mathcal{P}[(C_1, C_2, \ldots)] &\quad = \lambda\sigma.\mathcal{P}_k[C_1]\sigma \cup (\mathcal{P}[C_2, \ldots]\sigma) \\
\mathcal{P}[(\kappa, i)] &\quad = \lambda\sigma.\mathcal{P}_k[i]\sigma \rightarrow \sigma_i, \perp \\
\mathcal{P}[(i, c_j)] &\quad = \lambda\sigma.\mathcal{P}[c_j]\sigma \\
\mathcal{P}_k[\text{true}_c] &\quad = \lambda\sigma.\text{T} \\
\mathcal{P}_k[(T, \kappa, p, C_p)] &\quad = \lambda\sigma.\mathcal{E}[p]\mathcal{RS}(C_p, \sigma) \land \mathcal{P}_k[\kappa]\sigma \\
\mathcal{P}_k[(F, \kappa, p, C_p)] &\quad = \lambda\sigma.\neg\mathcal{E}[p]\mathcal{RS}(C_p, \sigma) \land \mathcal{P}_k[\kappa]\sigma \\
\end{align*}
\]

Auxiliary Functions:

\[
\begin{align*}
\mathcal{MS}(e, \gamma) &\quad = \{ (j, \gamma_j) \mid j \in \mathcal{R}(e) \} \\
\mathcal{MV}(p, \kappa, O, u, t, i, \gamma) &\quad = \{ (t, c^+(O, u, \kappa), p, \mathcal{MS}(p, \gamma), i, \mathcal{MS}(i, \gamma)) \} \\
\mathcal{RS}(C_e, \sigma) &\quad = \{ (j, \mathcal{P}[c_j]\sigma) \mid (j, c_j) \in C_e \} \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{c}^+(O, u, \text{true}_c) &\quad = \text{true}_c \\
\mathcal{c}^+(O, u, (\bar{u}, (\bar{t}, \bar{\kappa}, p, c_p))) &\quad = u \in \mathcal{O}(\bar{u}) \land u \not\in \bar{u} \Rightarrow \mathcal{c}^+(O, u, \bar{\kappa}, (t, \mathcal{c}^+(O, \bar{u}, \bar{\kappa}), p, c_p)) \\
\end{align*}
\]

Figure 6.10 PDT Function
This induction hypothesis is also admissible.

Case \((u)x := e\) The applicable denotations are as follows:

\[
C'[x := e]|O\bar{i}\bar{k}\bar{\gamma}\sigma = i \overset{?}{=} x \rightarrow \\
(ck'(O, u, \bar{k}, \sigma) \rightarrow E[e](\lambda j.\bar{\gamma}j\sigma)i, \bot), \bar{\gamma}i\sigma
\]

and

\[
P(G[(u)x := e]|O\bar{i}\bar{k}\gamma)\sigma = i \overset{?}{=} x \rightarrow \\
P((ck'^+(O, u, \kappa), e, MS(e, \gamma)))\sigma, \\
P(\gamma i)\sigma \\
= i \overset{?}{=} x \rightarrow \\
(P_k[ck'^+(O, u, \kappa)]\|\sigma \rightarrow \\
E[e]\mathcal{R}S(C_e, \sigma, \bot), P(\gamma i)\sigma \\
= i \overset{?}{=} x \rightarrow \\
(P_k[ck'^+(O, u, \kappa)]\|\sigma \rightarrow \\
E[e]\{(j, P[c_j]\|\sigma) | (j, c_j) \in C_e\}, \\
P(\gamma i)\sigma
\]

The theorem holds directly using the assumption on the inputs.

Case \((u)\text{if}(d_t, d_f, d_o) p s_t, s_f\) The applicable denotations are as follows:

\[
C'[\text{if}(d_t, d_f, d_o) p s_t, s_f]|O\bar{i}\bar{k}\bar{\gamma}\sigma = \\
i \notin d_t \cup d_f \rightarrow \bar{\gamma}i\sigma, \\
\text{let} \\
\bar{k}^+ = (u, \lambda \sigma.E[p](\lambda j.\bar{\gamma}j\sigma)) \circ \bar{k}, \\
\bar{k}^- = (u, \lambda \sigma.-E[p](\lambda j.\bar{\gamma}j\sigma)) \circ \bar{k}, \\
in \\
(i \notin d_t \rightarrow (E[p](\lambda j.\bar{\gamma}j\sigma) \land ck'(O, u, \bar{k}, \sigma) \rightarrow \\
\bar{\gamma}i\sigma, \bot), C'[s_t]|O\bar{i}\bar{k}^+(\lambda j\sigma'.j \notin d_o \rightarrow \bar{\gamma}j\sigma, \\
v_p'(T, p, \bar{\gamma}, \sigma, O, u, \kappa)\sigma' \rightarrow \bar{\gamma}j\sigma, \bot) \cup \\
(i \notin d_f \rightarrow (-E[p](\lambda j.\bar{\gamma}j\sigma) \land ck'^+(O, u, \bar{k}, \sigma) \rightarrow \\
\bar{\gamma}i\sigma, \bot), C'[s_f]|O\bar{i}\bar{k}^-(\lambda j\sigma'.j \notin d_o \rightarrow \bar{\gamma}j\sigma, \\
v_p'(F, p, \bar{\gamma}, \sigma, O, u, \kappa)\sigma' \rightarrow \bar{\gamma}j\sigma, \bot)
\]
and
\[
P(\mathcal{G}[\text{if}(d_i, d_j, d_o) \; p \; s_t \; s_f] \; Oi\kappa\gamma)\sigma = \\
i \notin d_i \cup d_j \rightarrow P(\gamma_i)\sigma,
\]
\[
P(\text{let } \kappa^+ = (u, (T, \kappa, p, MS(p, \gamma))), \\
\kappa^- = (u, (F, \kappa, p, MS(p, \gamma)))
\] in
\[
((i \notin d_i \rightarrow MV(p, \kappa, O, u, T, i, \gamma), \\
\mathcal{G}[s_t]Oi\kappa^+(\lambda j. j \notin d_o \rightarrow \gamma_j, MV(p, \kappa, O, u, T, j, \gamma))), \\
(i \notin d_f \rightarrow MV(p, \kappa, O, u, F, i, \gamma), \\
\mathcal{G}[s_f]Oi\kappa^-(\lambda j. j \notin d_o \rightarrow \gamma_j, MV(p, \kappa, O, u, F, j, \gamma))))\sigma
\]

If the identifier is not assigned in the body of the if statement, the theorem follows directly from the assumptions on the inputs. Otherwise, there are two cases to consider. If the identifier is assigned on both branches, the result follows from the induction hypothesis if the constructed control nodes \(\bar{\kappa}^+\) and \(\kappa^+\) yield the same value. The control nodes for the true branch are as follows:

\[
ck'(O, w, \bar{\kappa}^+, \sigma) = ck'(O, w, (u, \lambda \sigma. E[p](\lambda j. \bar{\gamma}j\sigma)) \circ \bar{\kappa}, \sigma) \\
= O(w, u) \rightarrow ck'(O, w, \bar{\kappa}, \sigma), \\
E[p](\lambda j. \bar{\gamma}j\sigma) \land ck'(O, u, \bar{\kappa}, \sigma)
\]

and
\[
P_k[ck^+(O, w, \kappa^+)]\sigma = P_k[ck^+(O, w, (u, (T, \kappa, p, MS(p, \gamma))))]\sigma \\
= O(w, u) \rightarrow P_k[ck^+(O, w, \kappa)]\sigma, \\
P_k[(T, ck^+(O, u, \kappa, p, MS(p, \gamma))]\sigma \\
= O(w, u) \rightarrow P_k[ck^+(O, w, \kappa)]\sigma, \\
E[p]RS(MS(p, \gamma), \sigma) \land \\
P_k[ck^+(O, u, \kappa)]\sigma
\]

If \(O(w, u)\) is true, the values are the same based on the input assumptions for the control parameters. Otherwise, the result holds using the store and control assumptions. The reasoning for the control nodes for the false branch is the same.

If the demanded identifier is assigned only on one branch, the input assumptions, the induction hypothesis, the information on the control nodes are all required. The reasoning follows the same pattern as in the proof for language \(W\).
Case $s_1; s_2$ This case requires the input assumptions and the induction hypothesis to establish that the input assumptions hold for the stores created for $s_2$. Then, the induction hypothesis establishes the theorem for the composition.

\[ \square \]

6.4 Operational Semantics in the Presence of GOTO

The \emph{pdt}s generated for programs with \texttt{goto} statements are the same as the previous \emph{pdt}s. Even though statements can have multiple control dependence predecessors, these statements are replicated in the \emph{pdt} such that each instance has only one control dependence predecessor. Therefore, the operational semantics is the same as that for language $\mathcal{W}$.

6.5 Example

This section presents a simple example of a program with \texttt{goto} statements but no loops. Section 7.4 presents examples of the text expansion function as it relates to loops. The example for this section is the following program:

\begin{align*}
x &:= e_1; \quad \quad \quad (1) \\
\text{if } p \quad & \quad \quad (2) \\
\text{then } & \quad \quad \text{goto 100} \quad (3) \\
\text{else} \quad & \quad \quad \text{goto 200} \quad (4) \\
100 : \quad x &:= e_2 \quad \quad (5) \\
200 : \quad y &:= f(x) \quad \quad (6) \\
\quad & \quad \quad \quad \quad \quad \quad z := g(y) \quad (7)
\end{align*}

The numbers at the end of each line are the statement numbers and are used for reference purposes in the discussion. The only statements that does not post-dominate the predicate in statement 2 are statements 1, which precedes the predicate, and
statement 5. The result of the text expansion is the following program:

\[
\begin{align*}
x & := e_1; \\
\text{if} & (\{x, y, z\}, \{y, z\}, \{x\}) \ p \quad (1) \\
\text{then} & \quad x := e_2; \quad (2) \\
& \quad y := f(x); \quad (5) \\
& \quad z := g(y) \quad (6) \\
\text{else} & \quad y := f(x); \quad (6) \\
& \quad z := g(y) \quad (7)
\end{align*}
\]

The \textit{pdt} for this program appears in Figure 6.11. The control dependences for the \textbf{start} node are left off as are the data predecessors for the two assignments to \(x\) and for the predicate \(p\), to simplify the picture. All of the node duplication is included.

The valve node for \(x\) comes from the check within the store that is sent to the branches of the \textbf{if} statement. The demand for \(z\) propagates into the \textbf{if} statement, and

![Figure 6.11  PDT Example with goto Statements](image)
then conceptually is removed from the if again with the control dependence filter. However, since the demand for \( z \) was sent to a branch of the if, the demands for the identifiers referenced by \( z \) also get sent to that branch, without checking to see if the identifier is assigned on the branch. The check in the store component captures this demand as it leaves the branch of the if statement and creates the appropriate valve node.

### 6.6 Discussion

For a usable theory of program dependence, general control flow must be considered. This chapter reduces the program of general control flow to that of structured control flow by exploiting the power of denotational definitions to cope with infinite text programs. While a naive expansion of the program results in too restrictive a control dependence relation, the appropriate relation is recovered using the post-dominator information from the unexpanded program. This information is found using the technique of abstract interpretation. Both the text expansion algorithm and the post-dominator algorithm are derived from the continuation-passing semantics of the goto language. The denotational definitions provide insight into the nature of control dependence in the context of general control flow.

Many efficient and useful compilation and optimization algorithms do not work on irreducible programs. Because the text expansion function performs the node splitting generally used to transform irreducible programs to reducible programs, the algorithms here do work with irreducible programs.
Chapter 7

Implementation Issues

The pdts described in the preceding chapters represent programs based on the data and control dependences present in the program; the more traditional formulations of the pdg represent programs using similar information but these representations must include pointers to the control flow graph or the original program text to maintain the meaning of the program. While pdts have a clean semantic specification, they pose a problem for compilers. Much of the technology that has been developed in the compiler area assumes a finite representation with access to the loop structure of the program. The infinite pdt, by unwinding the loops in the program, hides the loop structure of the program. Since most vectorizing and parallelizing transformations currently being implemented and studied focus on loops, this disruption in the program representation renders the pdt virtually useless for these applications.

However, the insights gained in the development of the pdt are applicable to the finite structures as well. This chapter describes how to apply these insights by presenting two algorithms. The first algorithm specifies the relationship between the finite and infinite program dependence structures. This algorithm collapses the infinite tree into a finite cyclic graph. The second algorithm transforms a program directly into a pdg with valve nodes. While the collapsing algorithm and the $G$ algorithms yield the same result, the second algorithm has the advantage of not requiring the creation of the infinite structure. In addition, it is more closely related to traditional compiler algorithms, so it should be easier to incorporate into existing compilers.

Section 7.1 describes how to collapse the pdt into a pdg. In Section 7.2, we discuss how this pdg corresponds to the pdgs generally given in the literature. The algorithm described in Section 7.3 generates the semantic pdg from the program text, without going through the pdt. For simplicity, we assume that data values are treated as atomic values. The modifications to the algorithm in Section 7.3 to accommodate arrays parallel those described in Chapter 4 to change the standard semantics to support arrays.
7.1 Collapsing PDTs to Semantic PDGs

The *pdt* described in the previous chapters are infinite trees. For a theory of dependence for *pdt* to apply to the structures used in compilers, the relationship between the finite graph and the infinite tree must be understood. This section describes a mapping function that maps the infinite tree for a program to the corresponding *pdg*. This *pdg*, however, still differs from that defined in Chapter 2. The *pdg* specified by the mapping function is called a *semantic program dependence graph*.

**Definition 7.1 (Semantic Program Dependence Graphs)** Let *Exp* be a set of expressions and *Id* a set of identifiers, deliberately unspecified. A semantic *pdg* $G = (N, E)$ is a potentially cyclic, finite, directed graph where

$$N = Id + (Id \times Exp) + Exp + Id + \{\text{start}\}$$

The node set can be partitioned into five disjoint sets: initial definition nodes, assignment nodes, predicate nodes, final nodes, and the designated start node. The edge multi-set can be partitioned into four sets: loop-dependent and loop-independent flow dependence and control dependence edges. Control dependence edges are labeled T or F. Control dependence edges must have either the start node or a predicate node as the source node; flow edges must have an assignment node as the source node. The start node and initial definition nodes have no incoming edges, and final nodes have no outgoing edges.

The correspondence between nodes in the tree and in the graph is found by associating tags with each node in the tree, corresponding to the statement in the program which the node represents; all nodes in the tree for a particular statement in general collapse to a single node in the graph. All *idef* or final nodes for a given identifier collapse into one node, and there is only one start predicate node. The correspondence between the edge sets, however, is more complicated. In addition, since valve nodes do not directly correspond to any program statement, they must be considered separately. In the infinite structure, edges from one loop iteration to another exist. In addition, one node may have edges to nodes in all iterations of a subsequent loop. Indeed, some nodes have edges to their corresponding node in the next loop iteration. These edge configurations must be distinguished from edges that exist between two nodes in the same iteration. Thus, we introduce the notion of loop-dependent dependences.
This concept is a generalization of the notion of loop-carried dependence described by Allen et al. [5]. While the definition of loop-carried dependence is sufficient for the purposes of vectorization and when a central store is present, the more general notion is needed to discriminate patterns of data flow in the finite pdg.

Definition 7.2 (Loop-Dependent Dependence) A dependence \( n \rightarrow m \) is \textit{loop-dependent} iff some path giving rise to the dependence includes the backward edge\(^{14}\) of a loop.

The following example program illustrates the distinction:

\[
\begin{align*}
y & := e_1; \\
x & := e_2; \\
\text{while } p(y) \\
y & := f(x, y)
\end{align*}
\]

The pdt for this program with \( y \) as the demanded identifier appears in Figure 7.1. To simplify the picture, some of the nodes have been combined within an iteration, but the nodes for the individual iterations are distinct. The pdt has flow edges from the assignment to \( x \) to each assignment to \( y \) within the loop. However, the assignment to \( y \) that is outside the loop has a flow edge only to the first assignment to \( y \) in the loop. These cases must be distinguished when the pdt is collapsed into a pdg. Since the assignment to \( x \) is not in a loop with the use of \( x \), the definition of loop-carried dependence [5] excludes this dependence from the class of loop-carried dependences. When collapsing a pdt into a pdg, edges are classified as either loop-independent or loop-dependent.

The collapsing algorithm appears in Figure 7.2. The nodes in each loop iteration are the same and the valve nodes prevent atomic values from flowing more than one iteration forward. Thus, all loop-dependent dependences that exist between any iterations will also exist between nodes in the first and the second iteration since there are no flow dependences that reach any further than a single iteration and dependence patterns do not change across iterations. Thus, the first step in the collapsing algorithm truncates the infinite tree at the second iteration for each loop. Because of the structure of the infinite tree after the text expansion, this truncation is

---

\(^{14}\)This definition is valid only in the context of reducible programs since irreducible programs do not admit the partitioning of the edge set into forward and backward edges.
Figure 7.1 Example PDT
main  truncate-tree
    label-tree(valve and loop header then its)
    make-node-set
    make-edge-set
end

label-tree
    label each loop-predicate with loop-level
    label each node with iteration number (0, 1 or 2)
    label loop-valve nodes with identifier, predicate label and iteration
    label non-loop valve nodes with identifier and predicate label
end

make-edge-set
    for each (x, y)
        if it(x) = it(y) ∧ loop(x) = loop(y) then
            add li(x, y)
        elseif it(x) + 1 = it(y) ∧ loop(x) = loop(y) then
            add ld(x, y)
        elseif loop(y) ≥ loop(x) ∧ it(y) = 1 then
            add li(x, y)
        elseif loop(y) ≥ loop(x) ∧ it(y) = 2 then
            add ld(x, y)
        elseif loop(x) ≥ loop(y) ∧ it(x) = 2 and x is loop-valve then
            add li(x, y)
    end

Figure 7.2  Collapsing Algorithm
performed simply by removing the third predicate node encountered for each predicate statement and all its successors.

The next step is to label the tree. This labeling identifies the loop predicates (headers), designates the iteration number for each node in a loop, and labels the valve nodes. Valve nodes that are control dependent on the loop-predicate are labeled uniquely across iterations; these nodes do not collapse to a single node for the loop. Instead, there are two valve nodes for each loop predicate: one for the value coming from before the loop and one for the values coming from with the loop. All other valve nodes are labeled with just the identifier and predicate label. Valve nodes in the loop body collapse to a single valve node.

The next step simply removes all duplicate nodes from the tree. Thus, all nodes in the second iteration of any loop are removed, except for the valve nodes. For each remaining loop-predicate node, there are two valve nodes.

Finally, the edge set must be reduced. Each edge from the truncated tree is examined. The processing is the same whether the edge is a control or flow dependence. If the edge is within a iteration for a loop, it is labeled as a loop-independent. If the edge goes from one iteration to the next in the same loop, it is a loop-dependent edge. If the edge goes into a loop to the first iteration, the edge is a loop-independent edge. If the edge goes into the loop to the second iteration, it is a loop-dependent edge. If the edge comes from the loop-dependent valve for a loop, the edge is loop-independent; this is the only way a value flows out from a loop.

Figure 7.3 contains the semantic pdg for the example program. There are two valve nodes for identifier y associated with the loop predicate. The first valve node has a loop-independent false control dependence; the second has a loop-dependent false control dependence. The flow edge from the assignment to y within the loop to the valve node is loop-dependent to pass the value out of the loop when the predicate evaluates to false. There are two flow edges from the assignment to x into the loop: a loop-independent and a loop-dependent flow dependence. The loop-independent edge represents the value flow to the first iteration; the second edge represents the value flow to all other iterations.

### 7.2 Characteristics of the Semantic Pdg

The semantic pdg specified by the collapsing algorithm differs from traditional pdgs defined in the literature. However, this pdg is closely related to the infinite pdt.
Indeed, the \textit{pdt} is the infinite unwinding of the semantic \textit{pdg}, following every possible path and creating distinct nodes in the tree for every node encountered in the graph. The presence of loop-dependent and loop-independent edges distinguishes between the data flow patterns in the example from the previous section for the traversal of loops during the unwinding process.

The differences between the semantic \textit{pdg} and traditional \textit{pdgs} arise from the use of valve nodes and loop-dependent dependences. The concept of loop-dependent dependences is a generalization of loop-carried dependences as described by Allen \textit{et al.} \cite{5}. Loop-carried dependences only exist between nodes that are in a common loop. In addition, for a loop-carried dependence to exist, the iteration of some common loop must give rise to the dependence. In the example from the previous section, there would only be a loop-independent dependence from the assignments outside the loop to the use of \textit{x} and \textit{y} inside the loop. The definition of loop-carried dependence is sufficient in the context in which it is applied: vectorizing and parallelizing transformations assuming a shared memory model of computation. However, this definition does not specify the flow of data into the loop, only the flow of data to the memory location. Thus, loop-carried dependence is insufficient in the context of valve nodes and a data flow model of computation that is independent of shared memory.
The semantic pdg contains valve nodes to control the flow of data. Valve nodes, while similar to the switching nodes in gated single assignment form [8] and the \( \phi \) nodes of static single assignment form [20], do not correspond exactly to other nodes serving this function. This question is explored in more detail in Section 8.2.1.

Using valve nodes to control the flow of data, no store component is required in the semantics except for the initial store which is the input to the program. In addition, no storage related dependences are necessary to maintain the meaning of the program. With valve nodes, control and flow dependences capture all of the necessary sequencing constraints on statement execution. This pdg also admits a compositional semantics.

### 7.3 Semantic PDG Algorithm

The semantic definitions described in this work rely on the valve node to control the flow of values arising from conditional assignments in a program. The valve node has a compositional semantics since it incorporates both the control information as well the data flow information required to select the appropriate value in the presence of conditional assignments. While the G algorithm, coupled with the collapsing algorithm described in the previous section, specifies where such nodes should be placed within a program, this algorithm relies on lazy implementations of infinite structures. In this section, we describe a relatively efficient algorithm, derived from the \( \mathcal{G} \) algorithm and the text expansion algorithm, that creates a semantic pdg with valve nodes in programs with general control flow. Figure 7.4 and Figure 7.5 present the modified text expansion algorithm and the modified \( \mathcal{G} \) algorithm respectively.

There are a few differences between this text expansion algorithm and the ones in the previous chapter. First, loops and labels are only expanded out for three iterations. The third iteration is required to get the valve nodes for the second iteration created properly. Creating the label environment no longer requires the fixed point operator. In addition, the while loop clause only expands out for three iterations; the help function \( f_3 \) performs this unwinding. The other change is the absence of the assignment sets. Since the program resulting from the expansion is always finite, the assignment function \( A \) from Chapter 3 properly computes the set of identifiers assigned in an if statement, so this information does not need to be computed by the text expansion function. As discussed in Section 7.1, only the first two complete iterations of any loop are required to compute the loop-dependent dependences in the
Semantic Domains:

\[
\begin{align*}
P & \in \text{pgm} \\
s, s_i & \in \text{stmt} \\
x & \in \text{ide} \\
e, p & \in \text{exp} \\
l & \in \text{lab} \\
u & \in \text{snun} \\
p & \in \text{labenv} = \text{lab} \rightarrow \text{stmt} \\
k & \in \text{cont} = \text{stmt} \oplus \epsilon
\end{align*}
\]

Semantics:

\[
T_p : \text{pgm} \rightarrow \text{pgm}
\]
\[
J : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{labenv}
\]
\[
T : \text{stmt} \rightarrow \text{labenv} \rightarrow \text{cont} \rightarrow \text{stmt}
\]

\[
T_p[e; (u)\text{end}] = T[e]r_3e; (u)\text{end}
\]
where \( r_i = J[e]r_{i-1}e \) for \( 1 \leq i \leq 3, r_0 = () \)

\[
\begin{align*}
A_T & : T[(u)x := e] = \lambda pk. k \hat{=} e \rightarrow (u)x := e, (u)x := e; k \\
I_T & : T[(u)if p \ s_t \ s_f] = \lambda pk. (u)if p (T[s_t]p \rho k) (T[s_f]p \rho k) \\
C_T & : T[s_1; s_2] = \lambda pk. T[s_1]p(T[s_2]p \rho k) \\
W_T & : T[(u)while p \ s] = \lambda pk. \text{unroll}(u, s, p, k) \\
G_T & : T[(u)\text{goto} \ l] = \lambda pk. pl \\
L_T & : T[l : s] = \lambda pk. T[s]p \rho k
\end{align*}
\]

\[
\begin{align*}
A_J & : J[(u)x := e] = \lambda pk. () \\
I_J & : J[(u)if p \ s_t \ s_f] = \lambda pk. \text{mkenv}(J[s_t]p \rho kd, J[s_f]p \rho kd) \\
C_J & : J[s_1; s_2] = \lambda pk. \text{mkenv}(J[s_1]p(T[s_2]p \rho k), J[s_2]p \rho k) \\
W_J & : J[(u)while p \ s] = \lambda pk. J[s]p(T[(u)while p \ s]p \rho k) \\
G_J & : J[(u)\text{goto} \ l] = \lambda pk. () \\
L_J & : J[l : s] = \lambda pk. \text{mkenv}(J[s]p \rho k, [l \leftarrow T[s]p \rho k])
\end{align*}
\]

where

\[
\text{unroll}(u, s, p, k) = (u)if p T[s]p((u)if p T[s]p((u)if p T[s]p \epsilon k) k) k
\]

**Figure 7.4** Finite Text Expansion Function Algorithm
semantic pdg. Thus, this finite unrolling of the program is sufficient to compute the finite structure.

The modified $G$ algorithm incorporates an additional parameter, $u_d$, which is the statement number of the statement initiating the demand. This statement number allows an additional filter on the creation of valve nodes that, while not affecting the meaning of the program, creates a cleaner semantic $pdg$. This demand filters out valve nodes for demands from nodes that are control dependent on the predicate of the valve node itself. The other change in this algorithm is the use of the function $\text{cons}^*$.\textsuperscript{15} This function simply maintains a list of nodes, referenced by statement number keys (as in the collapsing algorithm). The function $\text{cons}^*$ allows the creation of cyclic structures, and it reuses the nodes created on different paths in the program instead of duplicating the nodes. Thus, there is a single node for each statement in the program, and valve nodes as required both for while statements and if statements. The same filtering operation used in the collapsing algorithm partitions the edge sets resulting from this algorithm into the loop-dependent and loop-independent components.

The demand propagation correctly identifies the flow predecessors for the nodes in the graph, using the same mechanism as in the earlier chapters. The finite unrolling of the loops identifies all the loop dependences since the valve nodes prevent value flows from crossing multiple iterations. The if expansion of the while loop clearly demonstrates the placement of valve nodes for values flowing out of a loop. By tracking the source of the demand and by capturing demands as they flow out of the body of an if statement, valve nodes are created only when they are required, even in the presence of non-structured control. Finally, the algorithm above is reasonably efficient, as shown by the following theorem.

**Theorem 7.1** The semantic pdg algorithm creates a semantic $pdg$ for program $P$ in time $O(N^3)$ where $N$ is the number of statements in the program.

**Proof** The text expansion algorithm traverses each path in the program once for each pass, and there are four passes at most: at most three to find the label environment and one to expand the program, so the expansion takes $O(N^2)$. The filtering of the edge set takes a single pass over the edge set with each operation taking a constant amount of time;\textsuperscript{15}The function $\text{cons}^*$ is similar to the function $\text{hash-cons}$ in Pure Lisp.
Semantic Domains:

\[
\begin{align*}
v & \in \text{val}^T = \text{val} \cup \{T\} \\
i, x & \in \text{ide} \\
u & \in \text{snum} \\
O & \in \text{pdfun} = \text{snum} \rightarrow P_{fi}(\text{snum}) \\
e, p & \in \text{exp} \\
\sigma & \in \text{store}^T = \text{ide} \rightarrow \text{val}^T \\
\gamma & \in \text{code-tbl} = \text{ide} \rightarrow \text{snum} \rightarrow \text{code} \\
c & \in \text{code} = \text{d-node} \oplus (\text{d-node} \otimes \text{d-node}) \oplus \text{d-node}^+ \\
c_d & \in \text{d-node} = (\text{p-node} \otimes \text{exp} \otimes \text{code-tbl}) \oplus (\text{p-node} \otimes \text{ide}) \oplus (\text{p-node} \otimes \text{exp} \otimes \text{code-tbl} \otimes \text{code-tbl}) \oplus (\text{ide} \otimes \text{code}) \\
\kappa & \in \text{p-node} = \text{true}_c \oplus (\text{p-node} \otimes \text{exp} \otimes \text{code-tbl}) \oplus (\text{p-node} \otimes \text{exp} \otimes \text{code-tbl}) \oplus (\text{snum} \otimes \text{p-node})
\end{align*}
\]

Semantics:

\[
G : \text{stmt} \rightarrow \text{pdfun} \rightarrow \text{ide} \rightarrow \text{snum} \rightarrow \text{p-node} \rightarrow \text{code-tbl} \rightarrow \text{code}
\]

\[
\begin{align*}
G[u] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[u] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[if \ p \ s_1 \ s_f] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[if \ p \ s_1 \ s_f] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[s_1; s_2] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[s; (u)end] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
G[s; (u)end] & = \lambda \text{Oiudkyc} \cdot \gamma_i \xrightarrow{?} x \rightarrow \text{cons}^*((\text{ck}^+(O, u, \kappa), e, MS(e, \gamma, u))), \gamma_i u_d \\
\end{align*}
\]

Auxiliary Functions:

\[
\begin{align*}
\text{MS}(e, \gamma, u) & = \{(j, \gamma ju) \mid j \in R(e)\} \\
\text{MV}(p, \kappa, O, u, t, i, \gamma) & = \text{cons}^*((\text{cons}^*((t, \text{ck}^+(O, u, \kappa), p, \text{MS}(p, \gamma))), i, \text{MS}(i, \gamma))) \\
\text{ck}^+(O, u, \text{true}_c) & = \text{true}_c \\
\text{ck}^+(O, u, (\bar{u}, \text{cons}^*((t, \bar{\kappa}, p, c_p)))) & = u \in O(\bar{u}) \land u \neq \bar{u} \rightarrow \text{ck}^+(O, u, \bar{\kappa}), \\
\text{cons}^*((t, \text{ck}^+(O, \bar{u}, \bar{\kappa}), p, c_p)) & = u \in O(\bar{u}) \land u \neq \bar{u} \rightarrow \text{ck}^+(O, u, \bar{\kappa}), \\
\end{align*}
\]

Figure 7.5 Semantic PDG Function
it is thus bounded by the size of the set which is at most $O(N^2)$ with the pre-processing time required to label the nodes dominated by this time. The modified $G$ algorithm visits each path in a program at most once per identifier, assuming $cons^*$ does not recompute information that it has already obtained. Similarly, the cost of the $ck^+$ function is at most $O(N^3)$ amortized over the entire program, assuming that information is not recomputed, since each node can be checked at most once against all other paths of predicates. Since the number of identifiers is generally less than the number of statements in the program, the $G$ algorithm is bounded by $O(N^3)$. □

7.4 Examples of Semantic PDGs

This section presents examples illustrating the operations of the algorithms in the previous section. The first example is a program with no loops, but that does have goto statements. The second example contains a while loop; the third example is the same loop but this time created using a goto statement.

The first example is the example program from Section 6.5 in Chapter 6. The program is as follows:

\[
\begin{align*}
x &:= e_1; \\
\text{if } p & (2) \\
\text{then } & \text{goto 100} \\
\text{else } & \text{goto 200} \\
100: & \ x := e_2 \quad (5) \\
200: & \ y := f(x) \quad (6) \\
z &:= g(y) \quad (7)
\end{align*}
\]

The text expansion algorithm creates the same output as that found in Section 6.5. The label environments are identical, even though the fix operator is not used in the modified algorithm. The first pass of the label function suffices to find the continuations for the labels in this program. Because the demand for identifier $x$ post-dominates the predicate $p$ in the example, the check within the store built by the if clause creates the valve node for the false branch. Thus, the pdg created by this algorithm is the same as that shown in Figure 6.11 of the previous chapter.
The next two examples demonstrate the workings of the algorithm in the presence of loops. Both programs yield the same \textit{pdg} in the end as they both specify the same loop. The first program uses the \textbf{while} statement to specify the loop:

\begin{align*}
x & := e_1; & (1) \\
\text{while } p(x) & \quad (2) \\
x & := f(x); & (3) \\
y & := g(x) & (4)
\end{align*}

This loop expands into a series of three nested \textbf{if} statements as follows:

\begin{align*}
x & := e_1; & (1) \\
\textbf{if } p & \quad (2) \\
\textbf{then } x & := f(x); & (3) \\
\textbf{if } p & \quad (2) \\
\textbf{then } x & := f(x); & (3) \\
\textbf{if } p & \quad (2) \\
\textbf{then } x & := f(x) & (3) \\
\textbf{else } y & := g(x) & (4) \\
\textbf{else } y & := g(x) & (4) \\
\textbf{else} & \\
y & := g(x) & (4)
\end{align*}

The label environment for this program is empty since there are no labels specified in the program. The function \textit{cons} maps each of the instances of the predicate to the same node, as well as each of the instances of the assignments to \textit{x} and \textit{y} that are present in the \textbf{if} statements representing the loop. The \textit{pdg} for the program appears in Figure 7.6. The filter on the edge set distinguishes between the loop independent and loop-dependent control edges to the two valve nodes for the loop. The flow edges into the second valve node are loop-dependent since this value originates in the iteration just preceding the valve node. The assignment for \textit{x} within the loop has loop-dependent flow edges to itself and to the predicate node in addition to the loop-dependent flow edge to the valve node.

The same program can be constructed using a \textbf{goto} statement as follows:

\begin{align*}
x & := e_1; & (1) \\
100 & \textbf{if } p(x) & (2) \\
\textbf{then } x & := f(x); & (3) \\
\textbf{goto100} & & (4) \\
\textbf{else} & y & := g(x) & (5)
\end{align*}
The post-dominator relation for this program is clearly equivalent to that of the previous program. However, the label environment is not empty in this case. The label function iterates three times, expanding the if statement representing the loop. In the first environment $r_1$, the label 100 maps to the statement

$$\text{if } p x := f(x) \ y := g(x)$$

and in the second environment, it becomes

$$\text{if } p x := f(x); \text{if } p x := f(x) \ y := g(x) \ y := g(x)$$

The final iteration of the label function inserts the outermost if statement for the loop. Thus, the output of the text expansion function is the same as that from the while version of the program, and the pdg is obviously the same as well.
Chapter 8

Semantic Program Dependence Graphs

The $pdg$ is a convenient program representation that is appropriate for advanced optimizing and parallelizing compilers. The informal semantics ascribed to the $pdg$ is that of a data flow program, emphasizing the parallel, asynchronous nature of the representation. This nature is captured in the operational semantics for $pdts$ presented in the previous chapters. The parallel rewriting relations apply to nodes in the $pdt$ that have received all their relevant inputs. All nodes that are enabled for rewriting can be rewritten simultaneously. These semantic definitions show that the dependences in the $pdg$ are sufficient to capture the behavior of the program.

The denotational definitions developed in this dissertation provide a precise mathematical characterization of the different notions of dependence that are described in the literature and used in various program development systems. In addition, the staging analysis results in an algorithm that translates the text of a program into the $pdt$ for the program, and a corresponding interpreter for $pdts$. There is a mathematically precise relationship between $\mathcal{W}$, the original programming language semantics and $\mathcal{G}$ and $\mathcal{P}$, the compiler and interpreter for $pdts$.

The denotational analysis also provides insights into semantically sound, elegant solutions to some of the problems encountered when using dependence information. The outstanding example of this is the valve node. The valve node is a natural product of the staging transformation. However, the valve node addresses a problem for which no elegant solution had been found — the problem of multiple incoming values for a given identifier resulting from conditional assignments. The valve node resolves this problem by interrupting the flow of values around the conditional assignment if the conditionally assigned value is required. Thus, the valve node enforces the $dynamic$ constraint that only one value for each identifier flows into a node. Since the valve node contains the control dependences identifying the conditions under which the conditional assignment is not enabled, the valve node does have a compositional semantics. The meaning of each node depends only on the meaning of its components: its incoming edges with their respective values and its expression component.
The different properties of the valve node, the presence of the control information and the enforcement of the dynamic single assignment property, make the semantic \textit{pdg} an appropriate representation for optimization purposes. The compositionality of the semantics of this representation provides the basis for a semantic framework under which the validity of these transformations can be explored. The denotational semantics provides a mathematical basis for investigating the meaning of different optimizations. The operational semantics specifies an equational calculus for reasoning about the equality of \textit{pdt}s and \textit{pdgs} [50]. This calculus can also be used to determine the validity of optimizing transformations [49].

Chapter 7 presents a practical algorithm that places valve nodes in programs with general control flow. With this algorithm, the \textit{pdg} as proposed in this dissertation becomes a viable alternative to the \textit{ssa}-form \textit{pdg} or the other variants of the \textit{pdg} in use today. The resulting structure, the semantic \textit{pdg}, is an appropriate structure for use in compilers and programming environments, and has the additional advantage of the set of semantic tools mentioned above with which to prove properties about \textit{pdgs}, to understand the ramifications of extensions to \textit{pdgs} [39] and to reason about \textit{pdgs} and the operations on \textit{pdgs}.

The rest of this chapter first explores in more detail several issues raised throughout this dissertation. Section 8.2 recalls the problems with the \textit{pdg} as it was originally specified and surveys the different approaches that have been taken to remedy these shortcomings. Our approach is then compared with these approaches. The final section presents those areas in which further work is envisioned.

### 8.1 Issues in Program Dependence

#### 8.1.1 Store Filtering

The denotational definitions presented in this dissertation are based on the work of Cartwright and Felleisen [15]. They have proposed a modification to this work [14], which incorporates a masking function into the semantic functions. Two different masking functions, \textit{I} and \textit{M}, are introduced and used in conjunction with the three different store update functions discussed in Chapter 3. Under this formulation, the clauses in the semantics functions are not store transformers for the complete store. Instead, the clause for statement \textit{s} is a transformer for the portion of the store containing values for the identifiers assigned in the statement \textit{s}.
The semantic functions are parameterized over both the update functions and the masking functions. The $I$ masking function and the strict update function retain the strict semantics; the $M$ masking function is used with both the lackadaisical and lazy update functions. The same relationship holds among the resulting semantic functions as in Chapter 3:

$$W_{\text{strict}, I} \subseteq W_{\text{lack}, M} \subseteq W_{\text{lazy}, M}$$

The masking function,

$$M = \lambda \nu. \lambda \sigma. \lambda i. i \in \nu \rightarrow (\tau \sigma)(i), \sigma(i)$$

accepts a set of identifiers $\nu$ and a store transform function $\tau$ and restricts the effects of $\tau$ to the identifiers in the set $\nu$. The use of this masking function in conjunction with the lazy or lackadaisical update functions lifts the meaning of a program in the same way that the introduction of the identifier test in the clauses for the $\text{if}$ and $\text{while}$ statements for the semantic function $C$ lifts the meaning. If the demanded identifier is not assigned a value within the $\text{if}$ or $\text{while}$ statement, the store transformer for that statement does not enter into the computation of the demanded identifier.

For the array and $\text{abort}$ languages, there does not appear to be any complications in incorporating the store masking into the analysis performed in this dissertation. The situation is less clear for the $\text{goto}$ case since the text expansion alters the nesting structure of the program. We plan to incorporate the store masking functions into the extended functions presented here.

### 8.1.2 Lazy Semantics and Parallelism

The lazy semantics ascribed to various language constructs is not necessarily the expected semantics. However, since the goal of this analysis is to uncover parallelism and dependences restrict the degree of parallelism available in a program, the lazy semantics is the appropriate approach. For example, the lifting of the meaning of a program which occurs in the control stage of the derivation changes the meaning of a program by excluding from consideration a predicate that does not control the final value of the demanded identifier. Thus, the following program converges if $x$ is the demanded identifier:

$$\text{if } i/0 = 1 \quad y := 2 \quad y := 4; \quad x := 2$$
The predicate expression is treated as dead-code in this program, even though this treatment does alter the meaning of the program.

Another, perhaps less obvious example is the following program, once again with \( x \) as the demanded identifier:

\[
\text{if } p \text{ abort } y := 4; \quad x := 2
\]

Under the lazy interpretation of \texttt{abort}, this program denotes the value 2, not the abort value. Since the statements within the \texttt{if} statement do not directly influence the value of the demanded identifier \( x \), the entire \texttt{if} statement is disregarded. Thus, there is no control dependence for the assignment to \( x \) from the predicate \( p \), and the parallelism in the program increases as a result.

The lazy update function also restricts the dependence relations by substituting def-order dependences for, potentially many, output dependences. The following program illustrates the lifting of the meaning of a program when using the lazy update function, with \( x \) as the demanded identifier:

\[
x := \perp; \quad x := 2
\]

The first assignment to \( x \) is dead-code. However, disregarding this assignment causes this program to converge to a value instead of diverging.

In all of these examples, using the lazy interpretation of the language constructs decreases the number of dependences in the program and potentially lifts the meaning of the program from that specified by the traditional, sequential interpretation of the program. The lazy interpretation is useful for exploiting the additional parallelism in the program, but this additional parallelism requires the additional storage needed to support the implicit renaming of identifiers and the multiple versions of the arrays, denoted by the partial arrays. In addition, the optimized program, in some cases, looks radically different than the original program from the perspective of a debugger. These differences, as well as the changes in the program's meaning, complicate the process of debugging the program. The impact of the optimization process on the program's meaning and on the program itself should be understood and acknowledged by the compiler writer and the programmer.

\subsection{8.1.3 Lexical Scoping}

All identifiers and labels in the languages studied here have been global, and thus visible over the entire program; no scoping mechanism was included in any language
discussed here. In addition, the `abort` construct is a program level construct; the `exit`
and `continue` mechanisms for terminating loops were not discussed. For all of the
languages except the `goto` language, these extensions do not present any difficulties.
The situation, as discussed in the section on store masking above, is complicated by
the `goto` statements. The scoping decisions must be made before the text expansion,
or some provision must be made to retain the structure of the program in the output
of the text expansion since the program structure is significantly altered by the text
expansion function.

8.1.4 Procedures

The major limitation in the languages presented in this dissertation is the lack of
procedures. Historically, dependence analysis has focused on intra-procedural anal-
ysis, due to the complications inherent in analyzing information across procedure
boundaries, particularly when the modules are compiled separately. Recently, how-
ever, improved algorithms for handling inter-procedural analysis have been developed
(see, for example, Cooper et al. [18]).

Horwitz et al. [32] developed a dependence representation, called the `system de-
pendence graph` or `sdg`, which supports procedures with global variables and param-
eters passed by value-result. The `sdg` includes new nodes that handle the parameter
passing between the calling procedure and the called procedure. Veen [56] proposes a
procedure mechanism for data flow like languages. His representation is less cluttered
with additional nodes and edges than the `sdg`.

Assuming that the target of a `goto` statement must be in the same procedure as
the `goto` statement itself, incorporating procedures into the analysis is a straight-
forward process if all modules are available to the semantic functions, as in the case
where all modules are compiled together. One of the major problems with inter-
procedural analysis is the determination of alias information: when two names refer
to the same memory location. This problem is most apparent in the context of by-
reference parameter passing techniques and is of particular concern in the presence of
arrays since portions of arrays can be aliased to portions of other arrays. Cooper et
al. [18] and others have examined alias analysis in depth. Incorporating this analysis
can likely be accommodated into the semantic analysis using an oracle, as with the
subscript analysis in the array chapter, Chapter 4.
8.1.5 Pointers

The presence of pointers in a programming language significantly alters the practical analysis of dependence relations. While dependence is undecidable in general even without pointers, useful approximations are relatively easy to compute. In the presence of pointers, however, these algorithms are insufficient. In a programming language without types and with unrestricted pointer computations, a given pointer can conflict with any identifier in a program. Thus, a conservative computation of dependence must specify that this statement depends on all prior assignment statements. If pointer arithmetic is excluded or if types are available, some improvement on this dependence relation is possible. Horwitz, Pfeiffer and Reps [29], Pfeiffer [38], Gannon et al. [13, 26, 57] and others examine dependences and dependence-based representations in the presence of pointers, as well as methods of refining dependence relations in certain cases. Pfeiffer and Selke [39] establish the adequacy of dependence representations for programs accessing heaps. Given the problems with determining reasonable dependence relations in the presence of pointers, we have no plans to extend this work to accommodate pointer languages. Pfeiffer [38, 39] presents a reduction that maps any terminating program in a heap language to a program with only atomic identifiers. With this reduction, the analysis in this dissertation apply to programs using heap pointers.

8.1.6 Demand-Driven and Data-Driven Semantics

The denotational interpreter resulting from the staging analysis is a demand-driven interpreter. Not only is the analysis performed in a demand-driven fashion, as seen in the compiler function, but the interpreter only requests the evaluation of components that are actually needed to compute the final value for the demanded identifier. The following program illustrates the difference between a demand-driven interpreter and a data-driven interpreter, with $x$ as the demanded identifier:

$$z := 1/0; \quad \text{if } p \quad x := f(z) \quad x := 42$$

The denotation of this program is 42 if $p$ is true and $\bot$ otherwise. However, under the operational semantics for *pdts*, the meaning of this program is $\bot$, regardless of the outcome of the predicate $p$, since the computation of the value of $z$ is enabled for evaluation and thus must be evaluated before the computation of the program can complete. Until the value of $p$ is known, it is not known whether or not the value of
z is required to determine the value of x. If the predicate does denote true, however, there is no dependence on the assignment to z, and this expression computation should not affect the final result of the program.

This situation illustrates one of the issues in the distinction between static and dynamic dependence relations. The value of x clearly depends on the value of z under static analysis, because it is impossible to determine without the store what the value of p is. Once the value of p is known, however, we can refine the dependence relation to exclude the assignment to z.

In this example, a change to the termination criteria of the parallel operational semantics resolves this problem. The evaluation function currently requires all nodes to be resolved in the pdt. However, if the termination criteria instead only required a value to be present in the final node of the pdt, the behavior of the demand-driven interpreter is restored. Since the value nodes ensure that no more than one value arrives to any node for an identifier, if the final node has any value, it has the final value for that identifier. In effect, this change interrupts any computations that were begun but are no longer required. While this criteria restores the demand-driven character of the interpreter, to implement such a semantics in practice requires processor allocation to be performed at the level of the individual pdt nodes. Any coarser grain processor allocation would make this pre-emption impossible.

8.1.7 Differences between abort and goto

The interpretation attached to abort in this report differs from the traditional interpretation of abort. In general, abort is assumed to be equivalent to a goto statement with the end statement of the program as the target of the goto. The separate abort statement allows us to bypass the evaluation of predicate expressions if the demanded identifier is not defined in the body of the statement, even if an abort statement is present, in the same manner used for the structured control constructs. This change lifts the semantics and allows additional parallelism in the pdt. This same analysis is not performed on the goto statement. Indeed, unless a special check is inserted to determine if the target of the goto statement is the end label, this change is not legitimate. Thus, there is a fundamental difference between the abort construct as it is interpreted here and a goto statement to the end statement label.
8.1.8 Applicability to Optimizations

Since the goal of this work is to provide a semantic framework within which to analyze optimizations, the applicability of this work to those optimizations must be explored. As was discussed above, the optimizations are generally performed on a finite pdg, not on an infinite pdt. The mapping algorithm in the previous chapter allows the transfer of results from the infinite pdt to the finite pdg.

The denotational interpreter is a demand-driven interpreter yet most machines are data-driven machines. Thus, the termination criteria of the operational semantics models the execution strategy generally used on actual machines while the denotational interpreter reflects a more dynamic dependence structure. Reliance on dynamic dependence is not currently feasible in general, although the pre-emption strategy discussed in Section 8.1.6 above could be employed in some circumstances. Current execution strategies do not employ such pre-emption, so the differences in the semantics must be accommodated when applying the denotational results to actual compiler optimizations.

The granularity of the parallelism is also an issue. The denotational and operational semantics assume that each node in the pdt is allocated to an independent processor. In general, parallelizing compilers attempt to join code to increase the size of code that is allocated to a processor. Thus, a non-terminating computation can impact an otherwise unrelated computation if those computations are both allocated to the same processor. The loop-fusion optimization is an instance of where this combination of code occurs. Thus, although it seems obvious that combining the pdt for identifier x and the pdt for identifier y should not change the final values for these identifiers, this is not necessarily the case. If the value of x is ⊥ and the evaluation that leads to this divergence is allocated to a processor with a required part of the computation for the value of y, the result for y is also ⊥. Therefore, although the correctness of the loop fusion optimization can be established under the semantic definitions established here, the optimization is not valid under the normal processor allocation strategies. The role of processor allocation can not be ignored in the analysis of the optimizations.
8.2 Other Dependence-Based Graph Structures and Semantic Analyses of Optimization

The \( p_{dg} \) as it was originally defined presents the "partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program" [24, p.322]. The data flow model provides the informal semantics for the \( p_{dg} \), which represents the non-sequential semantics generally assigned to programs by compiler writers. However, Ferrante et al. did not present a semantics for the \( p_{dg} \). Horwitz et al. [31] were the first to address the theoretical foundations of \( p_{dgs} \), although the particular \( p_{dg} \) they use also differs from the original \( p_{dg} \). While the original \( p_{dg} \) uses output dependencies to sequence multiple assignments, the Horwitz et al. \( p_{dg} \) uses def-order dependences. There are actually two components to an output dependence. The first component sequences multiple definitions that may apply to a use of the identifier. The second component sequences distinct uses of the identifier name. While renaming of different uses of an identifier solves the second problem, the first problem remains. Horwitz et al. [31] address this problem by defining a def-order dependence to sequence multiple assignments that reach a common use of an identifier. The def-order dependence relation is a restriction of the transitive closure of the output dependence relation. Using the def-order relation provides more opportunities for parallelism than using the output dependence relation. Def-order dependences implicitly renames distinct uses of identifiers, but do nothing to address the problem of conditional assignments.

In trying to develop a data flow style semantics for \( p_{dgs} \) [48], it became clear that neither output dependence nor def-order dependence fit into the data flow model of computation. Our original graph rewriting semantics [48] did provide a semantics for the def-order \( p_{dg} \), but the semantics requires store components in each node in the \( p_{dg} \) to accommodate the potential over-writing of values in the case of multiple assignments for an identifier reaching a use of that identifier, violating the requirement for a compositional semantics. This is the most serious problem with the original \( p_{dg} \). Other minor problems were uncovered at later stages in the analysis. For example, to support a completely parallel semantics for \( p_{dgs} \), loop-dependent def-order dependences are required to allow different iterations of a loop to execute simultaneously. In addition, while the imprecise characterization of loop-carried dependences does not affect the uses that these dependences are put to in vectorization, this formulation
of the concept of loop dependence is insufficient to maintain the semantics of the program in a data flow setting.

There are approaches to addressing the problems of the original pdg other than that developed here. Cytron et al. [20] proposed static single assignment form to handle the problem of output dependence. A program in static single assignment form (ssa form) requires no static output dependences since there are never two assignments to the same variable in the program text and only one assignment reaches any use of an identifier. A program (pdg) in ssa form contains φ functions (φ nodes) that merge values from different sources for an identifier to maintain these invariants for program statements. Renaming is performed for all assignment statements so that no identifier has two assignment statements with the same identifier as the left-hand side. The φ functions, however, do not have a compositional semantics since the control information for the incoming data values is not given with the node. In addition, even though no static output dependences exist, there are dynamic output dependences between different loop iterations. Ssa form pdgs are used now in the program integration work of Horwitz et al., as well as in work on compiler optimization [6, 45, 59].

Horwitz, Reps, et al. use ssa form as the basis for their program representation graphs [61, 60], prgs, and system dependence graphs [32]. Both of these structures add φ nodes to the pdg corresponding to the φ functions in an ssa-form program. System dependence graphs also include interface nodes to support procedure calls. These structures still assume a structured program and scalar data values. Since the φ nodes are present, neither def-order nor output dependences are required. Ramalingam and Reps [44] provide a semantics for prgs as a system of equations, in the style of the original semantics for data flow programs, developed by Kosinski [34]. The φ functions prohibit the direct interpretation of the prg as a data flow graph since they do not correspond to any construct within the data flow model of computation. To overcome this, additional nodes are added to the graph to gate the values flowing into the φ nodes. These gating nodes perform the same function that valve nodes perform in pdts.

Beck et al. [11, 40, 41] have proposed a store-based data flow representation as an alternative to pdgs. Their representation, called the dependence flow graph, incorporates a global store into the data flow model of computation. They circulate access tokens to sequence access to the values in the store; an access token corresponds to a use of an identifier. These access tokens serve the function of def-order
or output dependence edges. Renaming occurs automatically in their translation, so anti-dependences are not necessary. The dependence flow graph supports general control flow; aliasing has been accounted for in the translation algorithm as well. Arrays are handled by creating access tokens for each entry in an array. This process is performed when a loop is discovered in the flow graph. The semantics presented for this structure [41, 40] forces loop iterations to run sequentially; this restriction significantly reduces the amount of parallelism available. An alternative proposal, bounding the number of active loop iterations [11], has not been explored further.

Ballance et al. [8] developed the program dependence web based on the ssa pdg. Their translation begins with the ssa pdg and transforms it into a gated single assignment form, or gsa, pdg. The \( \phi \) nodes in the ssa pdg map to gating sub-trees that identify the control conditions under which the different inputs to the \( \phi \) nodes are active. Nodes are also added to the gsa pdg to handle values flowing into and out of loops. This gsa pdg then translates to a data flow program by turning the gating nodes into data flow switches. The switches ensure that values do not flow into a region of code that does not execute under the given input conditions. One goal of this development is to find a structure that supports data-driven data flow interpretation, demand-driven data flow interpretation, and traditional sequential, control flow interpretation. The gating sub-trees provide the control information missing from the \( \phi \) nodes; thus, the sub-trees do have a compositional semantics although the representation as a whole does not have a compositional semantics, since the nodes used to merge the values from outside a loop with the values flowing from one iteration to the next do not have a compositional semantics.

Podgurski [43] analyzes several different dependence relations to determine which dependences are necessary to debug a program. The thesis is that dependences should specify which parts of the program contain the error in the program. This interpretation of dependence is different from that required for optimization. While the definitions overlap in some instances, the dependence relations presented in Podgurski are much broader.

Chapnerkar and Montenyohl [16, 17] used denotational semantics to explore the effects of different optimizations on the meaning of programs. The language under consideration is a structured language with only atomic data structure support. In addition, the expression language considered assumed the presence of only total functions over the integers. Thus, expressions could not return an undefined value. This work does not consider dependence-based representations or optimizations based on
these representations. Since the language considered is so restricted and no consideration is given to program dependence, this work does not address the problem of optimizations for parallelization in general scientific programs.

Aikens et al. [3] present a system for studying the validity of program transformations in the presence of multiple error values. They introduce an annotation, Safe, that identifies occurrences of functions that cannot produce errors. Using this annotation, they present a series of algebraic laws, within the language setting of FL, that support the application of program transformations, even in the presence of errors. These laws specify when these transformations do not change the error behavior of a program.

8.2.1 Comparison of the Different Representations

The valve node proposed in this dissertation has a compositional semantics, even in the presence of general control flow. The valve node is also useful in optimizations since it includes the information on when values flow from the node to the use of the value. This control information can be exploited by optimizations such as constant propagation. In addition, since valve nodes enforce a stronger property, dynamic single assignment, than ssa form, the same improvements in optimizations that are uncovered by ssa are also applicable to semantic pdgs.

Valve nodes are present in a pdt wherever def-order dependences exist. Although def-order dependences are better for optimization purposes than output dependences, def-order dependence is not a data flow concept. As shown in the operational semantics for def-order pdgs, some mechanism for saving and over-writing values must be present to accommodate def-order dependence. A mechanism for either storing or over-writing values would be required in a semantics for an ssa pdg. However, a semantics for valve nodes does not require any such mechanism, as demonstrated here. The valve nodes control the flow of values such that no node receives multiple values for a given identifier. Therefore, no over-rewriting of values is needed.

The dependence flow graph of Beck et al. [11, 40, 41] does not allow the evaluation of different iterations of a loop concurrently. Thus, this representation is completely unsuitable for use in a parallelizing compiler, since most such compilers specifically target loops as sources of potential parallelism. The goal of these compilers is to identify loops whose iterations can be run in parallel.
The \textit{gsa pdg} of Ballance \textit{et al.} \cite{9} is an interesting structure, in that it supports sequential, demand-driven and data-driven data flow interpretation. The gating nodes included in the \textit{gsa pdg} serve the same purpose as valve nodes, and have the same semantics. However, the placement of the switches does not correspond exactly to that of the valve nodes — there are situations where gating nodes appear but a valve node is not required. Thus, there is no precise semantic characterization of where these nodes should appear. In addition, the specific realization of the nodes controlling the flow of values into a loop and the values flowing from one iteration to the next in the \textit{gsa pdg} requires a state mechanism in the form of a first-time-through switch. This situation arises since loop behavior on the first iteration of a given instantiation is different than behavior on all subsequent iterations while a single loop input node in the \textit{gsa pdg} attempts to supply values for all iterations. Finally, the treatment of arrays in this setting is not sufficient to support the transformations required in advanced compilers.

\subsection{Future Work}

This dissertation describes a semantic foundation for optimization and parallelization using program dependences. The semantic analysis of dependence includes those types of dependence arising as a result of most imperative programming language features. Specifically, general control constructs and array structures have been included in the analysis of dependence, bringing the semantic analysis into the language realm where parallelization work is focused. The semantic framework allows the optimization process to be formalized. This formalization provides the language in which to characterize precisely the results of program transformations performed in current compilers.

There are several logical extensions to this work. The sections below describe the general areas in which extensions are planned: language enhancements, extending the semantics and the calculus, application to specific optimizations, and dynamic dependence.

\subsubsection{Language Enhancements}

First, the addition procedures is necessary to study interprocedural dependences. Several issues arise when procedures are added to the language. The parameter passing mechanism affects whether aliasing must be addressed. Aliasing occurs when
there are two names that refer to the same identifier. Aliasing is a particular problem with call-by-reference parameter passing. However, call-by-reference is not compatible with the data flow flavor of the \textit{pdt}. The data dependence relation is not altered significantly with the addition of procedures, although the dependences are in practice more difficult to uncover. Procedure calling requires additional control dependences linking the call site to the called procedure. Some mechanism for mapping argument values to their respective parameters in the procedure body is also needed. There do not appear to be any major problems with incorporating a procedure mechanism into the language, nor does there appear to any major insight to be gained from the addition.

\subsection{Additional Semantic Development and the Calculus}

Chapter 7 specifies a finite, semantic \textit{pdg} and shows how this structure relates to the \textit{pdt} that is the focus of this dissertation. However, this dissertation does not specify a semantics for this finite structure directly. The operational semantics originally developed for def-order \textit{pdgs} \cite{48} can be modified to provide such a semantics. The modified semantics must account for both general control flow and valve nodes.

In addition, the equational calculus for \textit{pdts} \cite{50} is defined only for \textit{pdts} that are images of structured programs with no special handling of array values. Extending the calculus to support array accesses and control dependence in the presence of non-structured control is necessary before the calculus can be applied to study the validity of most parallelizing transformations.

\subsection{Optimizations and Implementations}

The valve node addresses many of the same concerns as the $\phi$ nodes in \textit{ssa} form. The application of \textit{ssa} form to traditional optimization problems such as constant propagation \cite{59} has resulted in more efficient and more effective algorithms. In looking in particular at constant propagation, the valve node seems to allow for the same benefits as the \textit{ssa} form. In addition, there are constants that can be easily uncovered in the valve node \textit{pdg} that many current constant propagation algorithms do not find. The following program illustrates the problem:

$$ x := 6; \quad \text{if} \ x = 6 \ \text{then} \ x := 7 \ \text{else} \ y := 4; \quad z := f(x) $$

Clearly, the value of $x$ is 7 after the if statement. However, many algorithms miss this constant since the assignment $x := 6$ is not control dependent on the predicate.
With the valve node, however, the flow from this assignment is interrupted by the valve node. When the constant propagation algorithm determines that the predicate is always true, the flow from the valve node no longer applies, and the constant 7 is uncovered. There are other approaches to resolving this problem. The valve node structure, however, makes it obvious that this value is constant. In general, the additional control information present in the valve node construction provides the algorithm with the knowledge required to determine under what conditions particular paths are taken and thus what the value of an identifier must be. We plan to examine several optimizations in an attempt to improve the efficiency and effectiveness of these optimizations.

In conjunction with this work, we intend to use the equational calculus that is specified by the rewriting rules of the operational semantics to establish the validity of the different program transformations that implement these optimizations. The calculus must first be strengthened to allow for reasoning about additional data flow assumptions.

8.3.4 Dynamic Dependence

Traditional notions of program dependence, those discussed in this dissertation, have referred to static dependence. A static dependence exists if the order of the statements may affect the meaning of the program. Static dependence analysis algorithms examine the program text and consider all paths in determining dependences and make few, if any, decisions based on the actual values associated with identifiers in the program. However, dependence relations can change significantly during the execution of a program. Specifically, certain paths are not taken in a given run of the program, potentially changing the remaining dependence relations. In addition, information concerning the value of identifiers and expressions can refine the dependence relations further. These changes to the dependence relation can be exploited to increase parallelism and optimize program execution. We refer to this run-time notion of dependence as dynamic dependence.

Since the compiler community is responsible for most of the development occurring in the area of dependence analysis and their focus must clearly rest on static dependence, dynamic notions of dependence have not been fully explored.

Several fundamental question about dynamic dependence must be answered. First, a precise definition of dynamic dependence is needed. Current definitions address only
special cases of dynamic dependence, or are too vague. Next, the sources of dynamic
dependences should be explored. This work has established the link between certain
types of dependences and particular language usages or features. An understanding
of the situations in which dynamic dependences arise will facilitate the use of this in-
formation in optimization and parallelization. In addition, the relationship between
the execution model of computation and dynamic dependence needs to be explored,
as was discussed in Section 8.1.6 above, since the limitations of the pre-emption ap-
proach to dynamic dependences are not yet understood.

8.4 Semantics of Generalized Program Dependence

This dissertation uses the various tools of semantics to analyze the concept of pro-
gram dependence in the context of imperative programming languages. The ultimate
goal of this research is to provide a mathematical foundation for use in the optimiza-
tion and parallelization of scientific programs written in imperative languages. The
semantic pdg described in this work is a structure that could be exploited to just this
end. The semantic pdg retains the features of the traditional pdg upon which the
optimization process focuses — the loop structure — while still admitting a composi-
tional semantics. This structure can be generated reasonably efficiently and appears
to offer advantages to other structures in the design of optimizations.

The immediate goal of this research is to specify precisely the concept of program
dependence in imperative languages, to determine semantically what these depend-
dences mean and how they arise in a program, and to provide the basis for building
semantic tools to reason about and use the pdg. The different semantics described in
this dissertation characterize data and control dependence in imperative programs.
In addition, the different language features and usages are examined separately to
explore the impact of each feature on the overall dependence relations. These seman-
tic definitions not only provide the basis for other tools but also provide a simplified
mechanism for reasoning about pdgs and their properties. Thus, the tools of semantic
analysis can now be exploited in the domain of pdgs.
Bibliography


