The Semantics of Program Dependence

Robert Cartwright, Matthias Felleisen*
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Optimizing and parallelizing compilers for procedural languages rely on various forms of program dependence graphs (*pdgs*) to express the essential control and data dependences among atomic program operations. In this paper, we provide a semantic justification for this practice by deriving two different forms of program dependence graph—the *output pdg* and the *def-order pdg*—and their semantic definitions from *non-strict* generalizations of the denotational semantics of the programming language. In the process, we demonstrate that both the *output pdg* and the *def-order pdg* (with minor technical modifications) are conventional data-flow programs. In addition, we show that the semantics of the *def-order pdg* dominates the semantics of the *output pdg* and that both of these semantics dominate—rather than preserve—the semantics of sequential execution.

1 Program Dependence Graphs

Optimizing and parallelizing compilers for procedural languages rely on intermediate graph representations to express the essential control and data dependences of atomic program operations. Some prominent examples in the literature are the control flow graph, the call graph, the def-use chain [1], the data dependence graph [7], and the extended data-flow graph [2]. Recently, Ferrante, Ottenstein, and Warren [3] proposed a comprehensive graph representation called the *program dependence graph* (*pdg*) that is suitable for scalar optimization [3, 8], vectorization [14], and general parallelization [15]. Subsequently, Horwitz, Prins, and Reps [4] revised the *pdg* representation and used it as a basis for integrating program revisions. We refer to the original *pdg* [3] as the *output pdg* and to the revised form [4] as the *def-order pdg*.

A program dependence graph represents "[the] partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program" [3:322]. It expresses the informal non-sequential semantics that compiler writers assign to programs. The widespread use of the *pdg* representation as a basis for program compilation and optimization raises many interesting questions about the semantic properties of the analysis process.

- Does the program dependence graph representation have a "natural" interpretation as a data-flow programming language?
- What are the appropriate denotational and operational semantics for this language?
- Does the translation from a textual representation into a dependence graph representation preserve the program semantics?
- Does the semantics of *pdgs* justify the common optimizing transformations?
- Can a semantic characterization of program dependence provide the insight for better program optimization techniques?

As a first step towards a formal theory of program dependence, Horwitz, Prins, and Reps recently proved that the *def-order pdg* representation is *adequate*: programs with the same program dependence graphs have the same meaning. But no attempt has been made to provide a semantic justification for the

*The work of both authors was partially supported by NSF and DARPA.

program optimization process based on dependence analysis.

This paper, in conjunction with a companion paper by Selke [10], tackles the first three questions in the preceding list and presents three results:

1. The *output pdg* and *def-order pdg* representations [3, 4] for a program—with some minor technical modifications—are conventional data flow programs that can be *derived* from the denotational semantics of the original program.

2. The derivation also yields a denotational semantics of program dependence graphs that explains the significance of the data flow and control flow information.

3. The semantics of the *output pdg* is closer to the semantics of sequential execution than is the semantics of the *def-order pdg*. More precisely, the semantics of the *def-order pdg* dominates the semantics of the *output pdg*, and both dominate the semantics of the original programming language.

The remainder of the paper consists of four sections. The next section defines the syntax and conventional semantics for a simple procedural programming language and describes the *output pdg* and *def-order pdg* representations for this language. In Section 3, we analyze the semantics of this language from the perspective of program optimization and introduce two non-strict ("lazy") generalizations of the semantics that expose more parallelism. The fourth section contains our principal technical results. By semantically interpreting the concepts of control and data dependence, we derive a denotational definition of the control and data *demand* generated by program expressions. We then perform a staging analysis on the *demand semantics*, which produces a natural decomposition of the meaning function into two components: (i) a construction algorithm that builds program dependence trees and (ii) a meaning function that interprets these trees. At the end of Section 4, we describe how to build *pdgs* from program dependence trees. The final section presents directions for future research and some of the potential applications of our semantic characterization of program dependence.

2 The Programming Language W and Program Dependence

The typical programming language for optimizing and parallelizing compilers has a simple core consisting of assignment statements, sequencing constructs, loop constructs, and branching facilities. For the purpose of this discussion, we will focus our attention on a simple statement-oriented programming language **W** with the following abstract syntax:

$$\begin{aligned} s \;::=\; & x := e \mid \\ & s\,;\, s \mid \\ & \textbf{if } b \textbf{ then } s \textbf{ else } s \mid \\ & \textbf{while } b \textbf{ do } s \end{aligned}$$

where s ranges over statements, x ranges over a set of identifiers *ide*, and b and e range over a set of expressions *exp*. For the sake of generality, the set of expressions *exp* is left unspecified, but we assume that expressions may refer to identifiers in *ide*. The two functions $\mathcal{R}efs[\![\cdot]\!]$ and $\mathcal{D}efs[\![\cdot]\!]$ map programs and expressions to the sets of identifiers that occur in expressions and on the left-hand sides of assignment statements, respectively.

A definition of the denotational semantics of **W** is given in Figure 1. Since the syntactic domain *exp* is unspecified, the definition is parameterized over the domain *val* of denotations for expression and the meaning function $\mathcal{E}$ for mapping expressions to their denotations. The only restriction that we impose is the assumption that the domain *val* is *flat*, [1] and that it contains the domain $bool = \{\bot, \mathbf{T}, \mathbf{F}\}$ as a subset.

The semantic domain for **W**-programs is constructed from the domain of values as the domain of functions mapping stores to stores:

$$store \longrightarrow store\ .$$

A store is a mathematical representation of the machine's memory and maps identifiers to values:

$$ide \longrightarrow val\ .$$

The infix notation $\sigma[x \leftarrow v]$ represents a functional update of the store σ that is *strict* in all three arguments.[2] The updated store $\sigma[x \leftarrow v]$ is identical to σ except that $\sigma[x \leftarrow v]$ has the value v for variable x (assuming that σ, x, and v are all defined). Hence, the strict update function $\cdot[\cdot \leftarrow \cdot] : store \longrightarrow ide \longrightarrow val \longrightarrow store$ is defined by the following rules:

$$\sigma[x \leftarrow v] \;=\; \bot \quad \text{if } \sigma = \bot \vee x = \bot \vee v = \bot$$

and

$$\sigma[x \leftarrow v](z) \;=\; \begin{cases} \sigma(z) & z \neq x \\ v & z = x \end{cases}$$

[1] A *flat* domain **D** is a partially ordered set ordinary data values augmented by the undefined value bottom ($\bot$) denoting a divergent computation. The ordering relation $\sqsubseteq$ is the least relation such that every element approximates itself and the undefined value $\bot$ approximates every element.

[2] A function f is *strict* in an argument x iff it is undefined ($\bot$) for all inputs where x is undefined ($\bot$).

Semantic Domains:

$$\begin{array}{rcll} u, v & \in & val & (unspecified) \\ \sigma & \in & store = ide \longrightarrow val & \end{array}$$

Semantics:

$$\begin{array}{rcl} \mathcal{E} & : & exp \longrightarrow store \longrightarrow val \quad (unspecified) \\ \mathcal{W}_{strict} & : & stmt \longrightarrow store \longrightarrow store \end{array}$$

$$\begin{aligned} \mathcal{W}_{strict}[\![x := e]\!] &= \lambda\sigma.\sigma[x \leftarrow \mathcal{E}[\![e]\!]\sigma] \\ \mathcal{W}_{strict}[\![s_1 \,;\, s_2]\!] &= \lambda\sigma.\mathcal{W}_{strict}[\![s_2]\!](\mathcal{W}_{strict}[\![s_1]\!]\sigma) \\ \mathcal{W}_{strict}[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f]\!] &= \lambda\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow \mathcal{W}_{strict}[\![s_t]\!]\sigma, \mathcal{W}_{strict}[\![s_f]\!]\sigma \\ \mathcal{W}_{strict}[\![\textbf{while } b \textbf{ do } s]\!] &= fix(\lambda w\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow w(\mathcal{W}_{strict}[\![s]\!]\sigma), \sigma) \end{aligned}$$

Figure 1: The Denotational Semantics of **W**

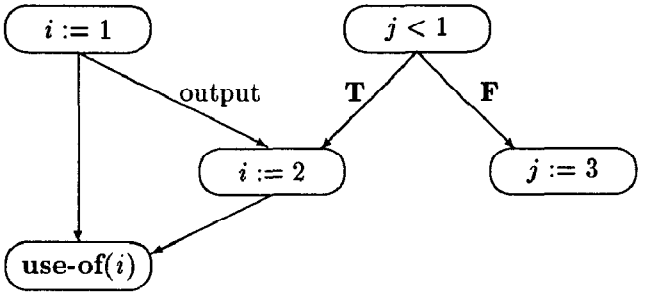

Figure 2: The *pdg* for program P

in all other cases.

A program dependence graph consists of a set of nodes representing atomic computations and a set of edges representing dependences. The atomic computations are assignment statements and predicate expressions. The set of edges contains *data* dependence edges and *control* dependence edges. A dependence edge between two nodes indicates that the computation of the *target* node depends on the computation of the *source* node.

A program node A that uses a variable v has a *data dependence* on a definition B of v iff there is a path in the flow chart representation of the program from B to A that does not contain an intervening definition of v. A predicate B *controls* a node A iff there are paths from B through the program that contain A and paths that do not contain A. B *directly controls* A if B controls A and there is a path from B to A that does not contain an intervening predicate that controls A. A program node A has a *control* dependence [3] on a predicate B iff B *directly controls* A.

There is another form of control dependence called *output* dependence [3] that must be respected for programs to execute in accord with the conventional sequential semantics. A definition A of a variable v has an *output* dependence on a preceding definition B of the same variable iff there is a path from A to B that does not contain an intervening definition of v. There is a relaxation of *output* dependence—more accurately, the transitive closure of *output* dependence—called *def-order* dependence [4] that ignores *output* dependences between definitions when there is no node that depends on both of them. A definition A of a variable v has a *def-order* dependence on a preceding definition B of the same variable iff there is a path from B to A and both A and B are the *sources* of data dependence edges to some program node.

The informal semantics underlying the dependence graph representation is an extended data-flow model of computation [3]. An assignment node in the graph computes a new identifier value; a predicate node determines a boolean value for the selection and elimination of other computation nodes. The incoming data dependence edges of a node specify the input values for the node's computation; the outgoing data edges indicate which nodes depend on the result of the node. Control dependences have the task of ordering, selecting, and eliminating computations.

The crucial property of the data-flow model of computation is the absence of a centralized store and store-oriented operations. There is no notion of overwriting values in locations. Instead, an assignment node directly passes the generated binding to its successor nodes in the graph.

As an illustration of the *pdg* representation, consider the program

$$\begin{aligned} P \stackrel{df}{=}\ & i := 1\,; \\ & \textbf{if } j < 1 \textbf{ then } i := 2 \textbf{ else } j := 3\,; \\ & \textbf{use-of}(i)\ . \end{aligned}$$

The translation of P into dependence graph form produces the *pdg* in Figure 2. There are two data-flow edges, one from each assignment to i to the node demanding the value of i. The predicate node $j < 1$ controls the second assignment to i through a **T**-control edge and the assignment to j through an **F**-control edge. An *output* or *def-order* edge—depending on the graph representation—between the two assignment nodes for i specifies the order of evaluation between the two assignments. Both truth-value-labeled edges and *def-order* edges denote control dependences. The representation of **while**-loops is similar to those of **if**-statements, but introduces cyclic dependences and requires a classification of data dependences as loop-dependent or loop-independent [3, 4].

3 Non-Strict Semantics for W

The denotational semantics for **W** reflects the behavior of a sequential implementation of **W**. An assignment statement alters the contents of a memory cell. The effect of a sequence of statements is the compound effect of evaluating its components from left to right. The **if**-statement selects the appropriate branch for further evaluation. A **while**-loop composes the effects of repeated evaluations of its body statement until its guard evaluates to **F**.

The sequentiality of program execution is reflected in the denotational semantics of **W** by the strictness of store update function $\cdot[\cdot \leftarrow \cdot]$ used in the semantic function $\mathcal{W}_{strict}$. If a statement s encountered in the sequential execution of the program p diverges for some input store σ, then the function $\mathcal{W}_{strict}[\![p]\!]$ applied to σ yields the completely undefined store $\perp_{store}$ regardless of what statements follow s. For example, the straight-line program

$$Q \stackrel{df}{=} y := 1\,;\ x := e\,;\ x := 2$$

diverges if the evaluation of the expression e diverges even though the value of e cannot affect the final value of x or y.

From the perspective of program optimization, the semantics of sequential execution is too restrictive. In the preceding program Q, the second assignment $(x := e)$ is superfluous because it can never contribute

Additional Semantic Domain:

$$\delta \in update = store \longrightarrow ide \longrightarrow val \longrightarrow store$$

Semantics:

$$\mathcal{W} : update \longrightarrow stmt \longrightarrow store \longrightarrow store$$

$$\begin{aligned}
\mathcal{W}\delta[\![x := e]\!] &= \lambda\sigma.\delta(\sigma, x, \mathcal{E}[\![e]\!]\sigma) \\
\mathcal{W}\delta[\![s_1 \,;\, s_2]\!] &= \lambda\sigma.\mathcal{W}\delta[\![s_2]\!](\mathcal{W}\delta[\![s_1]\!]\sigma) \\
\mathcal{W}\delta[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f]\!] &= \lambda\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow \mathcal{W}\delta[\![s_t]\!]\sigma, \mathcal{W}\delta[\![s_f]\!]\sigma \\
\mathcal{W}\delta[\![\textbf{while } b \textbf{ do } s]\!] &= fix(\lambda w\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow w(\mathcal{W}\delta[\![s]\!]\sigma), \sigma)
\end{aligned}$$

Figure 3: The Generalized Denotational Semantics of **W**

to a *final answer*. Fortunately, it is easy to modify the denotational definition of **W** so that no assignment is evaluated unless it can affect the final contents of the program store [9]. The trick is to replace the strict update operation $\cdot[\cdot \leftarrow \cdot]$ by an update operation that is *non-strict* in the third argument. There are two important forms of non-strict update:

1. The first option, denoted $\cdot[\cdot \Leftarrow \cdot]$, is the standard "lazy" generalization of $\cdot[\cdot \leftarrow \cdot]$. It is non-strict in both the first (store) and third (value) arguments. The store produced by the operation $\sigma[x \Leftarrow v]$ is defined if either x is defined and either σ or v is defined.[3] From an operational perspective, the computation of the value argument v is deferred until the value of the assigned identifier is demanded from the store. Moreover, the value v is never computed if a subsequent update on the same identifier is performed before the value is needed. Hence,

$$\sigma[x \Leftarrow v](z) = \begin{cases} \bot & x = \bot \\ \sigma(z) & z \neq x \wedge x \neq \bot \\ v & z = x \wedge z \neq \bot \end{cases}$$

   for all σ, x, v, z.

2. The second option, denoted $\cdot[\cdot \hookleftarrow \cdot]$, is partially "lazy". It is non-strict only in the third (value) argument because the operation $\sigma[x \hookleftarrow v]$ forces $\sigma(x)$ to be evaluated. Hence, updates to different variables cannot interfere with one another, but updates to the same variable can. In particular, $\sigma[x \hookleftarrow v]$ is defined at x only if both $\sigma(x)$ and v are defined. From an operational perspective, $\cdot[\cdot \hookleftarrow \cdot]$ treats $\bot$ as a "sticky" value: once a variable is bound to $\bot$ it will always be bound to $\bot$ regardless of subsequent updates. Hence, if $x \neq \bot$

$$\sigma[x \hookleftarrow v](z) = \begin{cases} \sigma(z) & z \neq x \\ v & z = x \wedge \sigma(x) \neq \bot \\ \bot & z = x \wedge \sigma(x) = \bot. \end{cases}$$

We refer to the update operations $\cdot[\cdot \Leftarrow \cdot]$ and $\cdot[\cdot \hookleftarrow \cdot]$ as the *lazy* update and the *lackadaisical* update, respectively. It is easy to verify that the lazy update operation dominates[4] the lackadaisical operation, and that both dominate the strict update operation:

$$\cdot[\cdot \leftarrow \cdot] \sqsubseteq \cdot[\cdot \hookleftarrow \cdot] \sqsubseteq \cdot[\cdot \Leftarrow \cdot]\,. \qquad (\dagger)$$

Given the two non-strict update functions, we can formulate two less restrictive semantic definitions for **W** and precisely describe their relation to the original semantics. To this end, we define a a new semantic function $\mathcal{W}$ based on $\mathcal{W}_{strict}$ that abstracts $\mathcal{W}_{strict}$

[3] There is one exception to this rule: the result is undefined if σ is defined only at x and v are undefined.

[4] A function $f : A \to B$ dominates a function $g : A \to B$ iff $\forall x \in A\ f(x) \sqsupseteq g(x)$: if the computation of $g(x)$ terminates, the computation of $f(x)$ must terminate and yield the same answer.

with respect to the update function. Figure 3 contains the complete definition of $\mathcal{W}$. From $\mathcal{W}$, we can obtain the original semantic function $\mathcal{W}_{strict}$ and the two non-strict variants as simple instantiations:

$$\begin{aligned} \mathcal{W}_{strict} &= \mathcal{W}(\cdot[\cdot \leftarrow \cdot]) \\ \mathcal{W}_{lack} &= \mathcal{W}(\cdot[\cdot \hookleftarrow \cdot]) \\ \mathcal{W}_{lazy} &= \mathcal{W}(\cdot[\cdot \Leftarrow \cdot]) \ . \end{aligned}$$

Since $\mathcal{W}$ is a continuous function of its update function argument, the approximation relations between update functions (†) imply the following approximation relations between the semantic functions:

$$\mathcal{W}_{strict} \sqsubseteq \mathcal{W}_{lack} \sqsubseteq \mathcal{W}_{lazy} \ .$$

The differences among these three semantics is best illustrated by an example. If the expression e in the program Q is undefined for all stores, then Q has a different meaning in each semantic definition:

$$\begin{aligned} \mathcal{W}_{strict}[\![Q]\!] &= \lambda\sigma.\bot_{store} \\ \mathcal{W}_{lack}[\![Q]\!] &= \lambda\sigma.\sigma(y) \stackrel{?}{\neq} \bot \longrightarrow \{(y,1)\} \\ \mathcal{W}_{lazy}[\![Q]\!] &= \lambda\sigma.\{(x,2),(y,1)\} \ . \end{aligned}$$

In short, while the lazy semantics ignores undefined values produced as the results of irrelevant subcomputations, the lackadaisical semantics only ignores the undefined values corresponding to irrelevant *variables*. Consequently, we interpret these two non-strict generalizations of the semantics of **W** as formalizations of two different perspectives on what assumptions an optimizing compiler should make. In the following section, we show how these two perspectives have influenced the design of program dependence representations.

4 A Semantic Approach to Dependence

The motivation behind translating programs into dependence graph representations is the conviction that the *functional representation* of a computation exposes potential parallelism hidden by the sequential, store-oriented evaluation model associated with the conventional program text [Warren: private communication]. On the other hand, the goal of denotational semantics [9] is to define a *functional meaning* for programs that explains their operational behavior, yet avoids machine-specific details.

The preceding observation suggests that denotational semantics is closely related to dependence analysis. For this reason, we decided to investigate the hypothesis that program dependence graphs are a form of parallel intermediate code that can be derived from a demand-oriented denotational semantics just as compilers and machine code can be derived from conventional denotational semantics [6, 11, 12, 13].

This section presents the results of that inquiry. In the first subsection, we show that a simple reformulation of a denotational definition exposes the data demands of expressions, revealing the data dependences between statements. The following subsection addresses the issue of control dependence. We introduce an explicit parameter for modeling the control predicates along the path from the root of the program to the current statement. The third subsection contains a staging analysis of the demand semantics. The resulting separation of the semantic function produces an algorithm for constructing dependence trees and a meaning function for these trees. The final subsection addresses the issue of how to generate program dependence graphs from our dependence trees.

4.1 Data Dependence

The primary goal of dependence analysis is to eliminate the notion of a centralized store and to illuminate the data flow between subcomputations within a program. To achieve this goal, we must answer two questions:

1. Which subcomputations are required to determine the value of an identifier at any given point in the program?
2. Where is each computed value used in the remainder of the program?

However, the semantic function $\mathcal{W}_{strict}$ and its instantiations are poorly suited to answering these questions because they presume a store-oriented model of computation. To eliminate this reliance on a central store, we must transform the semantic function $\mathcal{W}$ into a function $\mathcal{V}$ that extracts the value of a designated identifier—rather than the entire store—from a program.

A naïve way to write the new semantic function $\mathcal{V}$ would be to apply the result of $\mathcal{W}$ to an identifier. But this approach would merely hide the central store rather than eliminate it. Since we know the name of the designated identifier before the program is executed but know nothing about the initial store, the insight gained from research on deriving compilers from denotational semantics [6] suggests that we should rearrange the order of the arguments in $\mathcal{W}$ so that the identifier argument precedes the initial store argument. Then we can apply the transformed semantic function to the identifiers that appear in pro-

$$\mathcal{V} : update \longrightarrow stmt \longrightarrow ide \longrightarrow store \longrightarrow val$$

$$\begin{aligned}
\mathcal{V}\delta[\![x := e]\!] &= \lambda i\sigma.\delta(\sigma, x, \mathcal{E}[\![e]\!]\sigma)(i) \\
\mathcal{V}_{lazy}[\![x := e]\!] &= \mathcal{V}(\cdot[\cdot \Leftarrow \cdot])[\![x := e]\!] \\
&= \lambda i.i \stackrel{?}{=} x \longrightarrow \mathcal{E}[\![e]\!], \lambda\sigma.\sigma(i) \\
\mathcal{V}_{lack}[\![x := e]\!] &= \mathcal{V}(\cdot[\cdot \hookleftarrow \cdot])[\![x := e]\!] \\
&= \lambda i.i \stackrel{?}{=} x \longrightarrow (\lambda\sigma.\sigma(i) \stackrel{?}{=} \bot \longrightarrow \bot, \mathcal{E}[\![e]\!]), \lambda\sigma.\sigma(i) \\
\mathcal{V}\delta[\![s_1 \,;\, s_2]\!] &= \lambda i\sigma.\mathcal{V}\delta[\![s_2]\!]i(\lambda j.\mathcal{V}\delta[\![s_1]\!]j\sigma) \\
\mathcal{V}\delta[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f]\!] &= \lambda i\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow \mathcal{V}\delta[\![s_t]\!]i\sigma, \mathcal{V}\delta[\![s_f]\!]i\sigma \\
\mathcal{V}\delta[\![\textbf{while } b \textbf{ do } s]\!] &= fix(\lambda w.\lambda i\sigma.\mathcal{E}[\![b]\!]\sigma \longrightarrow wi(\lambda j.\mathcal{V}\delta[\![s]\!]j\sigma), \sigma(i))
\end{aligned}$$

Figure 4: The Generalized Demand Semantics of **W**

gram expressions as part of a static analysis process akin to compilation. Hence, we want to rewrite the semantic function

$$\mathcal{W} : update \longrightarrow stmt \longrightarrow store \longrightarrow (ide \longrightarrow val)$$

as a semantic function

$$\mathcal{V} : update \longrightarrow stmt \longrightarrow ide \longrightarrow store \longrightarrow val$$

that is mathematically equivalent, *i.e.*,

$$\mathcal{V}\delta sx\sigma = \mathcal{W}\delta s\sigma x \quad \text{for all } \delta, s, x, \sigma \ .$$

The complete definition of $\mathcal{V}$ is given in Figure 4. Like $\mathcal{W}$, $\mathcal{V}$ can be instantiated using three different update functions to produce three different, but closely related, semantic functions:

$$\begin{aligned}
\mathcal{V}_{strict} &= \mathcal{V}(\cdot[\cdot \leftarrow \cdot]) \\
\mathcal{V}_{lack} &= \mathcal{V}(\cdot[\cdot \hookleftarrow \cdot]) \\
\mathcal{V}_{lazy} &= \mathcal{V}(\cdot[\cdot \Leftarrow \cdot]) \ .
\end{aligned}$$

A closer examination of the non-strict versions of $\mathcal{V}$ reveals that we can simplify the definition of their assignment clauses. In both definitions, we can move the equality test on the identifier forward, out of the update operation. For the lazy update, this analysis produces the following assignment clause:

$$\begin{aligned}
\mathcal{V}_{lazy}[\![x := e]\!] &= \lambda i\sigma.\sigma[x \Leftarrow \mathcal{E}[\![e]\!]\sigma](i) \\
&= \lambda i\sigma.i \stackrel{?}{=} x \longrightarrow \mathcal{E}[\![e]\!]\sigma, \sigma(i) \\
&\qquad \text{(by the definition of } \cdot[\cdot \Leftarrow \cdot]) \\
&= \lambda i.i \stackrel{?}{=} x \longrightarrow \mathcal{E}[\![e]\!], \lambda\sigma.\sigma(i) \ .
\end{aligned}$$

For the lackadaisical update, the same analysis yields a slightly different result:

$$\mathcal{V}_{lack}[\![x := e]\!] = \lambda i.i \stackrel{?}{=} x \longrightarrow (\lambda\sigma.\sigma(i) \stackrel{?}{=} \bot \longrightarrow \bot, \mathcal{E}[\![e]\!]), \lambda\sigma.\sigma(i) \ .$$

These two transformations show that it is possible to identify the final assignments for a given identifier by statically analyzing the denotational meaning of the program—provided the update functions are non-strict. But this form of analysis alone is too weak to identify the data dependences in a program. For a given identifier, the semantic functions $\mathcal{V}_{lazy}$ and $\mathcal{V}_{lack}$ identify the final assignment(s) to the identifier and then wait for the computation of the input store. Without the store, it is not clear how to determine the denotation of the right-hand side of the assignment statement.

At this point, we must recall the second question from the beginning of this subsection: how do we determine where a value is needed? Before the semantic function $\mathcal{V}$ can produce the final value for some identifier x, we must provide the data values for evaluating the right-hand sides of the final assignments for

x. Since the syntactic function $\mathcal{R}efs[\![\cdot]\!]$ determines the finite set of identifiers referenced in an expression, we can restrict the data demand of an expression e to the values of all identifiers in the set $\mathcal{R}efs[\![e]\!]$ instead of demanding the entire store. In other words, the right-hand side of a final assignment generates a demand for the values of a finite set of identifiers. The definition of $\mathcal{V}$ tells us how to propagate this demand. Since the input store σ to each final assignment is built by the recursive application of the function $\mathcal{V}$, the application of σ to the demanded identifiers reveals the final assignments to these identifiers within the preceding computation. This process can be repeated to identify all of the data dependences in the program. In fact, we could perform a staging analysis as in Section 4.3 to generate a "structural" form of program dependence graph similar to the *pdgs* presented in [2]. However, this form of *pdg* expresses data dependences between expressions and compound statements instead of between expressions and assignments. To analyze data dependence at the level of atomic assignments and predicates, we need to analyze control information as well.

4.2 Control Dependence

The second component of program dependence analysis is the determination of control dependences between assignments and predicates. In the definition of the semantic function $\mathcal{V}$, the code for handling **if** and **while** is motivated by an operational interpretation of **W**. After evaluating the predicate, the function selects the appropriate branch and determines the value of the demanded identifier computed by that branch. However, from a mathematical perspective, the function determines the values of *both* branches and selects the relevant one afterwards. The value generated by the non-selected branch is often bottom ($\perp_{val}$) because the predicates typically enforce conditions that prevent the execution of erroneous or irrelevant computations. This observation suggests a strategy for reformulating $\mathcal{V}$: if the revised semantic function produces the value $\perp$ for the non-selected branches, then it can simply *merge* the results of alternative statements.

The obvious choice for the merge operation is the *least upper bound* operation $\sqcup$, which on a flat domain always returns the defined value when applied to a defined value and bottom. Unfortunately, the $\sqcup$ operation is not defined on the *entire* domain *val*, because the least upper bound of two non-bottom values does not exist. Although this situation cannot occur within **W**-programs, we need to extend the *val* domain to cast the least upper bound operation as a continuous function on the entire domain. We can accomplish this task by attaching a top element $\top$ to the domain *val* above all of the other elements. The resulting domain $val^{\top}$ then forms a complete lattice and the least upper bound operation is a continuous function. An added benefit of this extension is that it creates a framework in which we can define the semantics of program dependence graphs that are *not* the images of **W**-programs.

To incorporate $val^{\top}$ in our revised denotational definition of **W**, we must also modify the definition of the domains *store* and *bool*, which depend on *val*. We will use the domains

$$store^{\top} = ide \longrightarrow val^{\top}$$

and

$$bool^{\top} = bool \cup \{\top\}$$

instead of the domains *store* and *bool*.

Given the extended domain of values $val^{\top}$, we can now abstract $\mathcal{V}$ with respect to an explicit *control parameter* (κ) that represents the accumulated boolean value of the predicates along the control path from the root of the program to the current statement. This parameter passes information to atomic assignments indicating whether or not they are reachable in a given computation. In the former case, they pass their computed values to all of the sites (assignments and predicates) that demand them. In the latter case, they pass the default value $\perp$. When the values from all of the final assignments to an identifier in a program component are merged to produce the demanded answer, only one of the final assignments is reachable. Without this extra control information, we cannot transmit data values directly from definition sites (assignments) to their use sites (assignments and predicates). The control parameter directly corresponds to the control information that is passed along control dependence edges in a *pdg*.

The abstraction $\mathcal{V}$ with respect to the control parameter κ yields a new function $\mathcal{C}$ with the type

$$\begin{aligned} update \quad &\longrightarrow \quad stmt \longrightarrow ide \longrightarrow bool^{\top} \longrightarrow store^{\top} \\ &\longrightarrow \quad val^{\top} \ . \end{aligned}$$

A complete definition of $\mathcal{C}$ including instantiations for the non-strict updates $\cdot[\cdot \Leftarrow \cdot]$ and $\cdot[\cdot \hookleftarrow \cdot]$ appears in Figure 5. The clause defining the meaning of assignments is:

$$\mathcal{C}\delta[\![x := e]\!] \quad = \quad \lambda i \kappa \sigma . \kappa \longrightarrow \delta(\sigma, x, \mathcal{E}[\![e]\!]\sigma)(i), \perp \ .$$

When the update operation δ is instantiated as $\cdot[\cdot \Leftarrow \cdot]$ or $\cdot[\cdot \hookleftarrow \cdot]$, we can unfold the definition of the

Modified Semantic Domains:

$$\begin{array}{rcll} u, v & \in & val^{\top} & (val \cup \{\top\}) \\ \sigma & \in & store^{\top} = ide \longrightarrow val^{\top} & \end{array}$$

Semantics:

$$\mathcal{C} : update \longrightarrow stmt \longrightarrow ide \longrightarrow bool \longrightarrow store^{\top} \longrightarrow val^{\top}$$

$$\begin{array}{rcl} \mathcal{C}\delta[\![x := e]\!] & = & \lambda i\kappa\sigma.\kappa \longrightarrow \delta(\sigma, x, \mathcal{E}[\![e]\!]\sigma)(i), \bot \\ \mathcal{C}_{lazy}[\![x := e]\!] & = & \mathcal{C}(\cdot[\cdot \Leftarrow \cdot])[\![x := e]\!] \\ & = & \lambda i\kappa\sigma.i \stackrel{?}{=} x \longrightarrow (\kappa \longrightarrow \mathcal{E}[\![e]\!]\sigma, \bot), \sigma(i) \\ \mathcal{C}_{lack}[\![x := e]\!] & = & \mathcal{C}(\cdot[\cdot \hookleftarrow \cdot])[\![x := e]\!] \\ & = & \lambda i\kappa\sigma.i \stackrel{?}{=} x \longrightarrow (\kappa \wedge \sigma(i) \neq \bot \longrightarrow \mathcal{E}[\![e]\!]\sigma, \bot), \sigma(i) \\ \mathcal{C}\delta[\![s_1 \,;\, s_2]\!] & = & \lambda i\kappa\sigma.\mathcal{C}\delta[\![s_2]\!]i\kappa(\lambda j.\mathcal{C}\delta[\![s_1]\!]j\kappa\sigma) \\ \mathcal{C}\delta[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f]\!] & = & \lambda i\kappa\sigma.i \notin \mathcal{D}efs[\![s_t]\!] \cup \mathcal{D}efs[\![s_f]\!] \longrightarrow \sigma(i), \\ & & \quad \textbf{let } \kappa^+ = \kappa \wedge \mathcal{E}[\![b]\!]\sigma;\ \kappa^- = \kappa \wedge \neg\mathcal{E}[\![b]\!]\sigma \\ & & \quad \textbf{in } (i \notin \mathcal{D}efs[\![s_t]\!] \longrightarrow (\kappa^+ \longrightarrow \sigma(i), \bot), \mathcal{C}\delta[\![s_t]\!]i\,\kappa^+\,\sigma) \sqcup \\ & & \qquad (i \notin \mathcal{D}efs[\![s_f]\!] \longrightarrow (\kappa^- \longrightarrow \sigma(i), \bot), \mathcal{C}\delta[\![s_f]\!]i\,\kappa^-\,\sigma) \\ \mathcal{C}\delta[\![\textbf{while } b \textbf{ do } s]\!] & = & fix(\lambda w.\lambda i\,\kappa\,\sigma. \\ & & \quad i \notin \mathcal{D}efs[\![s]\!] \longrightarrow \sigma(i), \\ & & \quad \textbf{let } \kappa^+ = \kappa \wedge \mathcal{E}[\![b]\!]\sigma;\ \kappa^- = \kappa \wedge \neg\mathcal{E}[\![b]\!]\sigma \\ & & \quad \textbf{in } (wi\kappa^+(\lambda j.\mathcal{C}\delta[\![s]\!]j\kappa^+\sigma)) \sqcup (\kappa^- \longrightarrow \sigma(i), \bot) \end{array}$$

Figure 5: Demand and Control Semantics of **W**

update operation to reveal both the control dependences and the data dependences between statements and expressions.

The definition of $\mathcal{C}$ includes one important feature that does not occur in the definition of $\mathcal{V}$. In particular, the clauses in $\mathcal{C}$ defining the meaning of **if** and **while** statements interpret these statements as identity transformations (empty statements) if they are *irrelevant* (no definition of the demanded variable occurs in the text of the statement). This feature *changes the meaning* of **W** because it eliminates a possible source of divergence, namely the evaluation of the predicates governing the execution of irrelevant **if** and **while** statements. Hence, $\mathcal{C}$ satisfies the following two conditions

$$\mathcal{V}\delta[\![p]\!]i\sigma \;\sqsubseteq\; \mathcal{C}\delta[\![p]\!]i\mathbf{T}\sigma$$

$$\bot \;=\; \mathcal{C}\delta[\![p]\!]i\mathbf{F}(\lambda j.\bot)\ .$$

We could eliminate this difference between $\mathcal{V}$ and $\mathcal{C}$ at the cost of generating extra control dependences on the predicates of irrelevant **if** and **while** statements. In this case, $\mathcal{C}$ would satisfy the stronger condition

$$\mathcal{V}\delta[\![p]\!]i\sigma \;=\; \mathcal{C}\delta[\![p]\!]i\mathbf{T}\sigma$$

instead of the inequality relation given above.

4.3 Program Dependence Trees

Now that we have produced a denotational semantics that identifies the essential data and control dependences for computing the value of each identifier, we can perform a staging analysis to extract a code representation for **W**-programs [6, 11, 12, 13]. A staging

$$\mathcal{C}' : update \longrightarrow stmt \longrightarrow ide \longrightarrow (store^{\top} \longrightarrow bool^{\top}) \longrightarrow (ide \longrightarrow store^{\top} \longrightarrow val^{\top}) \longrightarrow store^{\top} \longrightarrow val^{\top}$$

$$\begin{aligned}
\mathcal{C}'\delta[\![x := e]\!] &= \lambda i\kappa\gamma\sigma.\kappa\sigma \longrightarrow (\delta(\sigma, x, e))(i), \gamma i\sigma \\
\mathcal{C}'_{lazy}[\![x := e]\!] &= \mathcal{C}'(\cdot[\cdot \Leftarrow \cdot])[\![x := e]\!] \\
&= \lambda i\kappa\gamma.i \stackrel{?}{=} x \longrightarrow \lambda\sigma.(\kappa\sigma \longrightarrow \mathcal{E}[\![e]\!](\lambda j.\gamma j\sigma), \bot), \lambda\sigma.\gamma i\sigma \\
\mathcal{C}'_{lack}[\![x := e]\!] &= \mathcal{C}'(\cdot[\cdot \hookleftarrow \cdot])[\![x := e]\!] \\
&= \lambda i\kappa\gamma.i \stackrel{?}{=} x \longrightarrow \lambda\sigma.(\kappa\sigma \wedge \sigma(i) \stackrel{?}{\neq} \bot \longrightarrow \mathcal{E}[\![e]\!](\lambda j.\gamma j\sigma), \bot), \lambda\sigma.\gamma i\sigma \\
\mathcal{C}'\delta[\![s_1 \,;\, s_2]\!] &= \lambda i\kappa\gamma.\mathcal{C}'\delta[\![s_2]\!]i\kappa(\lambda j.\mathcal{C}'\delta[\![s_1]\!]j\kappa\gamma) \\
\mathcal{C}'\delta[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_f]\!] &= \lambda i\kappa\gamma. \\
&\quad i \notin \mathcal{D}efs[\![s_t]\!] \cup \mathcal{D}efs[\![s_f]\!] \longrightarrow \gamma i, \\
&\quad \textbf{let } \kappa^{+} = \lambda\sigma.\kappa\sigma \wedge \mathcal{E}[\![b]\!](\lambda j.\gamma j\sigma);\ \kappa^{-} = \lambda\sigma.\kappa\sigma \wedge \neg\mathcal{E}[\![b]\!](\lambda j.\gamma j\sigma) \\
&\quad \textbf{in } (i \notin \mathcal{D}efs[\![s_t]\!] \longrightarrow (\lambda\sigma.\kappa^{+}\sigma \longrightarrow \gamma i\sigma, \bot), (\lambda\sigma.\mathcal{C}'\delta[\![s_t]\!]i\kappa^{+}\gamma\sigma)) \sqcup \\
&\qquad (i \notin \mathcal{D}efs[\![s_f]\!] \longrightarrow (\lambda\sigma.\kappa^{-}\sigma \longrightarrow \gamma i\sigma, \bot), (\lambda\sigma.\mathcal{C}'\delta[\![s_f]\!]i\ \kappa^{-}\ \gamma\sigma)) \\
\mathcal{C}'\delta[\![\textbf{while } b \textbf{ do } s]\!] &= fix(\lambda w.\lambda i\kappa\gamma. \\
&\quad i \notin \mathcal{D}efs[\![s]\!] \longrightarrow \gamma i, \\
&\quad \textbf{let } \kappa^{+} = \lambda\sigma.\kappa \wedge \mathcal{E}[\![b]\!](\lambda j.\gamma j\sigma);\ \kappa^{-} = \lambda\sigma.\kappa \wedge \neg\mathcal{E}[\![b]\!](\lambda j.\gamma j\sigma) \\
&\quad \textbf{in } (\lambda\sigma.\kappa^{-}\sigma \longrightarrow \gamma i\sigma, \bot) \sqcup (wi\kappa^{+}(\lambda j.\mathcal{C}'\delta[\![s]\!]j\kappa^{+}\gamma))
\end{aligned}$$

Figure 6: Staged Semantics of **W**

analysis decomposes a denotational definition into a static meaning function, which constructs an intermediate representation for a program, and an interpreter, which executes the intermediate code. A critical preliminary step in a staging analysis is to transform the definition of the meaning function to make as many phrases as possible independent of the program input. Then the static meaning function can construct concrete representations for the results of applying these phrases to program text.

In our case, the staging analysis must determine which parameters of the semantic function $\mathcal{C}$ depend on the initial store. We can then construct a new semantic function $\mathcal{C}'$ that abstracts the affected parameters with respect to the initial store and passes the initial store as the final argument. From the specification of $\mathcal{C}$'s domain

$$stmt \longrightarrow ide \longrightarrow bool^{\top} \longrightarrow store^{\top} \longrightarrow val^{\top}$$

and its semantic clauses, it is clear that at an arbitrary point in the evaluation, both the control parameter and the store parameter depend on the initial store. This observation suggests that function $\mathcal{C}'$ should have the the type

$$\begin{aligned}
stmt &\longrightarrow ide \longrightarrow (store^{\top} \longrightarrow bool^{\top}) \\
&\longrightarrow (store^{\top} \longrightarrow store^{\top}) \\
&\longrightarrow store^{\top} \longrightarrow val^{\top}.
\end{aligned}$$

But this formulation of $\mathcal{C}'$ treats the program text preceding the current statement as a function from $store^{\top}$ to $store^{\top}$. As we noted earlier, the meaning of the current statement does not depend on the entire input store, but only on the values of selected variables. Hence, we can re-apply the same trick that we used to eliminate the central store from $\mathcal{W}$: move the identifier argument from the answer store (produced in this case by the preceding program text) to the beginning of the argument list yielding the type

$$\begin{aligned} stmt &\longrightarrow ide \longrightarrow (store^{\top} \longrightarrow bool^{\top}) \\ &\longrightarrow (ide \longrightarrow store^{\top} \longrightarrow val^{\top}) \\ &\longrightarrow store^{\top} \longrightarrow val^{\top}. \end{aligned}$$

In writing new semantic function $\mathcal{C}'$, we must preserve the semantics given by the function $\mathcal{C}$. The initial control parameter $\mathbf{T}$ and the initial store of $\mathcal{C}$ become parameters that are abstracted over the initial store, and the initial store becomes the last argument:

$$\mathcal{C}'\delta[\![p]\!]i(\lambda\sigma.\mathbf{T})(\lambda i\sigma.\sigma(i))\sigma_0 \quad = \quad \mathcal{C}\delta[\![f]\!]pi\mathbf{T}\sigma_0.$$

Given this invariant, it is straightforward to rewrite $\mathcal{C}$'s clauses to match the types of the new semantic domain.

The result of the staging analysis appears in Figure 6. This definition shows that it is possible to determine which decisions made during conventional program execution are independent of the initial store and where the initial store is really needed. More importantly, the evaluation of a program using the function $\mathcal{C}'$ does not rely on a central store; the only stores involved are local stores for each assignment statement and predicate expression.[5]

The last step in our staging analysis is the decomposition of $\mathcal{C}'$ into two functions: a static meaning function and an interpreter. If we apply $\mathcal{C}'$ to all the required arguments except the initial store, it returns a function that maps the initial store to a value. But this function has exactly the same type as abstract machine code, which maps initial stores to answers. If we devise a concrete representation for this code, we can decompose $\mathcal{C}'$ into a "compiler" function $\mathcal{G}$ that constructs the code and an "interpreter" function $\mathcal{P}$ that determines the meaning of code.

The separation of the semantic function $\mathcal{C}'$ into the functions $\mathcal{G}$ and $\mathcal{P}$ is a simple process governed by the invariant

$$\begin{aligned} &\mathcal{P}(\mathcal{G}(\cdot[\cdot \Leftarrow \cdot])[\![p]\!]i(\mathbf{true}_c)(\lambda i.\langle i, \{\}, \{\}\rangle))\sigma_0 \\ &\quad = \quad \mathcal{C}'(\cdot[\cdot \Leftarrow \cdot])[\![p]\!]i(\lambda\sigma.\mathbf{T})(\lambda i\sigma.\sigma(i))\sigma_0. \end{aligned}$$

To accomplish this separation, $\mathcal{G}$ must construct a concrete representation every time $\mathcal{C}'$ returns a function that maps the initial store to a value. For example, when $\mathcal{C}'(\cdot[\cdot \Leftarrow \cdot])$ returns the function

$$\lambda\sigma.(\kappa\sigma \longrightarrow \mathcal{E}[\![e]\!](\lambda j.\gamma j\sigma), \bot)$$

in the assignment clause, the parameters of this definition are the expression e, the control parameter κ, and the intermediate function γ that maps the identifiers in e to the code trees for computing the respective values. Hence, the code representation for this function is the triple

$$\langle e, \{(j, \gamma j) \mid j \in Refs[\![e]\!]\}, \kappa\rangle$$

since the rest of the function is reconstructible from this information. The other clauses of $\mathcal{G}$ are derived in a similar way.

The semantic function $\mathcal{P}$ reconstructs the denotations of $\mathcal{C}'$ from the code produced by $\mathcal{G}$. The finite set formation operation (denoted by braces $\{\cdot\}$) in the definition of $\mathcal{P}_c$ is *strict*. Consequently, a code expression c is never evaluated until *all* of its data dependences and control dependences have been satisfied.

Figure 7 defines the concrete code representation and the functions $\mathcal{G}$ and $\mathcal{P}$. These definitions rely on standard domain constructions and operators explained in the Appendix. An assignment statement $x := e$ produces a *data-node* consisting of the uninterpreted right-hand side expression e, a finite set of dependence trees indexed by the identifiers in e and a *control-node*. A *control-node* is either the atomic code $\mathbf{true}_c$, which represents the initial predicate, or a *data-node* tagged with the value **T** or **F**; the expression field in the tagged *data-node* is the text of the controlling predicate and the tag indicates the boolean value that the predicate must produce to satisfy the control dependence. An **if**-statement generates a pair of *data-nodes*, one for each arm of the **if**-statement. A **while**-loop produces an infinite sequence $[d_0, d_1, \ldots, d_k, \ldots]$ of *data-nodes*, where d_k computes exactly k iterations of the loop.

The trees that we generate for the *lazy* semantics and the *lackadaisical* semantics, respectively, are essentially the infinite unwindings of the corresponding *def-order pdgs* and *output pdgs*. The only difference between the *lazy* code trees and the infinite unwindings of *def-order pdgs* is the presence of *valve* nodes in the code trees instead of *def-order* edges.[6] Similarly, the only difference between our *lackadaisical* code trees and the infinite unwindings of *output pdgs* is the presence of valve nodes in our code trees in addition to the *output* dependence edges.

[5] By making more assumptions about the expression language, we could construct graph representations for expressions and eliminate the local stores.

[6] There are two other minor differences between our code trees and *def-order pdgs* as defined by Horwitz *et al.* Their version of the *pdg* is designed to support program integration instead of program optimization. Consequently, they include nodes corresponding to dead code because dead code may become live after subsequent integration steps. In addition, they omit *loop-carried def-order* edges, which apparently are unnecessary for program integration.

Semantic Domains:

$$
\begin{array}{rcll}
c_\sigma \in \textit{code-table} & = & \textit{ide} \multimap \textit{code} & \text{(finite table of code)} \\
\textit{data-node} & = & \textit{exp} \otimes \textit{code-table} \otimes \textit{control-node} & \text{(data node: } \langle e, c_\sigma, c_\kappa \rangle\text{)} \\
c \in \textit{code} & = & \textit{data-node} \oplus (\textit{data-node} \otimes \textit{data-node}) \oplus & \text{(code)} \\
 & & \quad \textit{data-node}^+ & \\
c_\kappa \in \textit{control-node} & = & \mathbf{true}_c \oplus \textit{data-node} \oplus \textit{data-node} & \text{(control node:} \\
 & & & \quad \mathbf{true}_c, \langle \mathbf{T}, b, c_\sigma, c_k \rangle, \langle \mathbf{F}, b, c_\sigma, c_k \rangle\text{)}
\end{array}
$$

Generating Dependence Trees:

$$\mathcal{G} : \textit{update} \longrightarrow \textit{stmt} \longrightarrow \textit{ide} \longrightarrow \textit{control-node} \longrightarrow \textit{code-table} \longrightarrow \textit{code}$$

$$\mathcal{G}_{lazy}[\![x := e]\!] = \lambda i\kappa\gamma.i \stackrel{?}{=} x \longrightarrow \langle e, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![e]\!]\}, \kappa \rangle, \gamma i$$

$$\mathcal{G}_{lack}[\![x := e]\!] = \lambda i\kappa\gamma.i \stackrel{?}{=} x \longrightarrow \langle e, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![e]\!]\} \cup \{(i, \gamma i)\}, \kappa \rangle, \gamma i$$

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

$$
\begin{array}{rcl}
\mathcal{G}\delta[\![\textbf{if } b \textbf{ then } s_t \textbf{ else } s_e]\!] & = & \lambda i\kappa\gamma. \\
 & & \quad i \notin \mathcal{D}efs[\![s_t]\!] \cup \mathcal{D}efs[\![s_f]\!] \longrightarrow \gamma i, \\
 & & \quad \textbf{let } \kappa^+ = \langle \mathbf{T}, b, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![b]\!]\}, \kappa \rangle; \\
 & & \quad\quad \kappa^- = \langle \mathbf{F}, b, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![b]\!]\}, \kappa \rangle \\
 & & \quad \textbf{in } \langle (i \notin \mathcal{D}efs[\![s_t]\!] \longrightarrow \langle \langle i, \{(i, \gamma i)\}, \kappa^+ \rangle \rangle, \mathcal{G}\delta[\![s_t]\!]i\kappa^+\gamma)), \\
 & & \quad\quad (i \notin \mathcal{D}efs[\![s_f]\!] \longrightarrow \langle \langle i, \{(i, \gamma i)\}, \kappa^- \rangle \rangle, \mathcal{G}\delta[\![s_f]\!]i\kappa^-\gamma)) \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G}\delta[\![\textbf{while } b \textbf{ do } s]\!] & = & \mathit{fix}(\lambda w.\lambda i\kappa\gamma. \\
 & & \quad i \notin \mathcal{D}efs[\![s]\!] \longrightarrow \gamma i, \\
 & & \quad \textbf{let } \kappa^+ = \langle \mathbf{T}, b, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![b]\!]\}, \kappa \rangle; \\
 & & \quad\quad \kappa^- = \langle \mathbf{F}, b, \{(j, \gamma j) \mid j \in \mathit{Refs}[\![b]\!]\}, \kappa \rangle \\
 & & \quad \textbf{in } \langle i, \{(i, \gamma i)\}, \kappa^- \rangle \circ (wi\kappa^+(\lambda j.\mathcal{G}\delta[\![s]\!]j\kappa^+\gamma))
\end{array}
$$

Interpreting Dependence Trees:

$$\mathcal{P} : \textit{code} \longrightarrow \textit{store}^\top \longrightarrow \textit{val}^\top$$

$$
\begin{array}{rcl}
\mathcal{P}[\![[c_1, c_2, \ldots]]\!] & = & \lambda\sigma.\mathcal{P}[\![c_1]\!]\sigma \sqcup \mathcal{P}[\![[c_2, \ldots]]\!]\sigma \\
\mathcal{P}[\![\langle c_1, c_2 \rangle]\!] & = & \lambda\sigma.\mathcal{P}[\![c_1]\!]\sigma \sqcup \mathcal{P}[\![c_2]\!]\sigma \\
\mathcal{P}[\![\langle e, c_e, c_k \rangle]\!] & = & \lambda\sigma.\mathcal{P}_c[\![c_k]\!]\sigma \longrightarrow \mathcal{E}[\![e]\!]\{(j, \mathcal{P}[\![c_j]\!]\sigma) \mid (j, c_j) \in c_e\}, \bot
\end{array}
$$

$$\mathcal{P}_c : \textit{control-node} \longrightarrow \textit{store}^\top \longrightarrow \textit{bool}^\top$$

$$
\begin{array}{rcl}
\mathcal{P}_c[\![\mathbf{true}_c]\!] & = & \lambda\sigma.\mathbf{T} \\
\mathcal{P}_c[\![\langle \mathbf{T}, b, c_e, c_k \rangle]\!] & = & \lambda\sigma.\mathcal{P}_c[\![c_k]\!]\sigma \wedge \ \mathcal{E}[\![b]\!]\{\langle j, \mathcal{P}[\![c_j]\!]\sigma \rangle \mid (j, c_j) \in c_e\} \\
\mathcal{P}_c[\![\langle \mathbf{F}, b, c_e, c_k \rangle]\!] & = & \lambda\sigma.\mathcal{P}_c[\![c_k]\!]\sigma \wedge \neg\mathcal{E}[\![b]\!]\{\langle j, \mathcal{P}[\![c_j]\!]\sigma \rangle \mid (j, c_j) \in c_e\}
\end{array}
$$

Figure 7: Dependence Tree Semantics of **W**

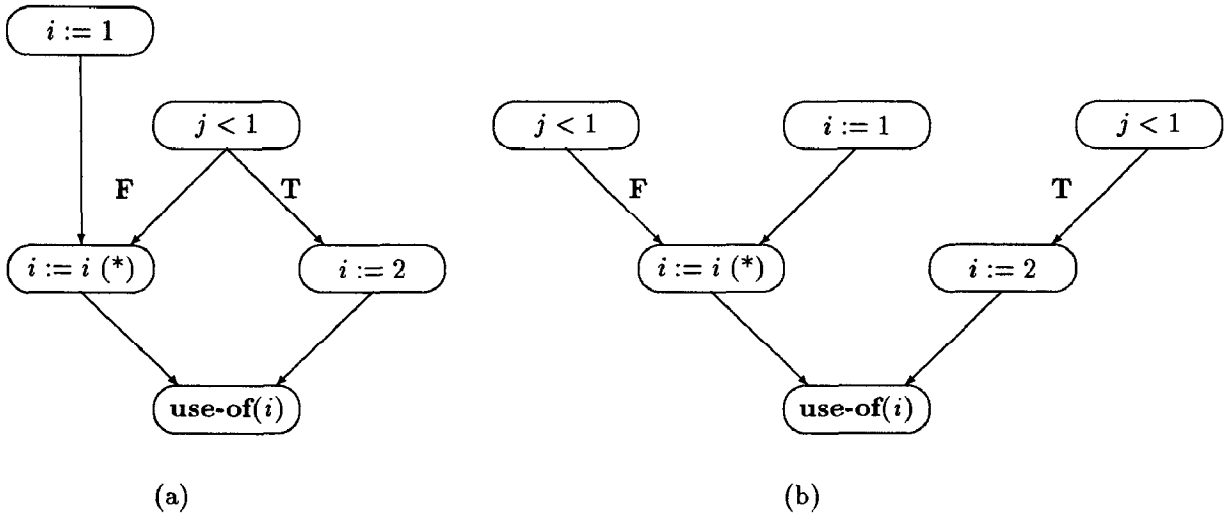

Figure 8: Semantic *pdg* for Program P

Valve nodes are produced by the code generator γ when it discovers the definition (final assignment) of a demanded identifier in **while**-loop or in only one arm of a **if**-statement. The valve controls the flow of the value produced by the *preceding* definition to the demanding site. If the new definition of the identifier is executed then the valve blocks the flow of the value from the preceding definition. As we mentioned above in the discussion of the semantic function $\mathcal{C}$, the valve nodes ensure that only one non-bottom value is transmitted to a consuming node. From a semantic perspective, the *def-order* and *output* edges included in conventional *pdg* representations are insufficient. Neither form of edge passes the precedence information about multiple definitions of the same identifier to the consuming nodes that must make the discrimination. Thus, in a conventional *def-order* or *output pdg*, a consuming node may receive several different values for the same identifier, but it cannot determine which one is "correct" without access to global information about the status of the associated *output* or *def-order* dependences.

Figure 8 illustrates the difference between our code trees and the unwound *def-order pdgs* for the simple program P (presented as a conventional *pdg* in Figure 2). When the use-node for i has received the value from the first assignment to i and the second assignment has yet to provide a value, the use-node cannot tell whether to accept the first value for i as the final one or to wait for a value to be transmitted along the second data-flow edge for i. The decision depends on the evaluation of the *def-order* or *output* edge between the first and second assignment to i, which is completely disassociated from the use-node for i. The introduction of *valve* nodes as in Figure 8 solves this problem. Since the new valve node (*) is under the control of the **if**-predicate and causes either the second assignment node or the valve node to send the value $\perp_{val}$, only one defined value ever reaches the use-node. As a result, the evaluation of the use-node no longer depends on non-associated pieces of the graph.

4.4 Collapsing Trees into Graphs

To collapse a tree into a graph, we partition the nodes of the tree into a finite set of equivalence classes and construct the graph determined by merging all of the nodes in the same equivalence class. The equivalence relation is defined by equating all of the tree nodes that correspond to the same assignment statement or predicate in the program. Valve nodes are equated if they have have the same controlling predicate (in the program text), the same demanded variable, and the same truth label (**T** or **F**) on the edge to the controlling node.

For loop-free programs, the unwinding of the collapsed graph is identical to the original code tree because all of the nodes in an equivalence class in the tree have exactly the same predecessors (subtrees): see Figure 8. For programs with loops, however, the collapsed graph does not contain enough information to reconstruct the original code tree. The problem is that we cannot distinguish *loop-*

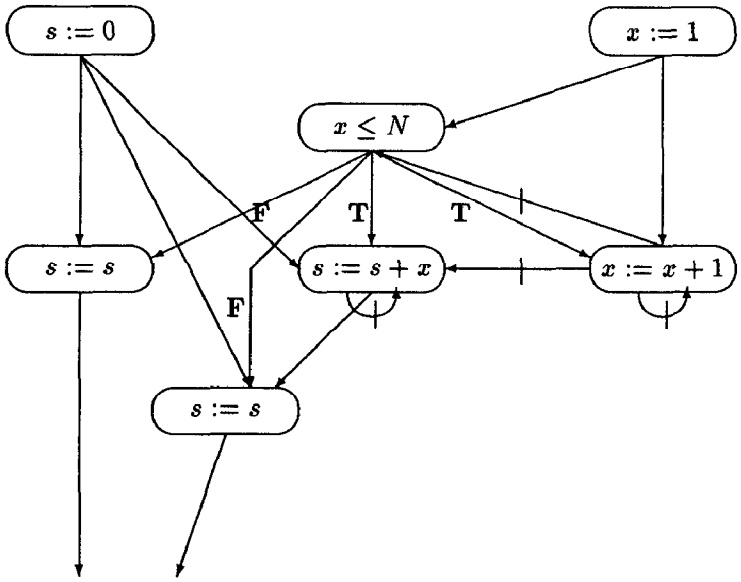

Figure 9: Semantic-based Pdg for Program R

carried data dependence edges from *loop independent* edges. We can overcome this problem by appropriately labeling the edges involving loop nodes, but the details of the tree reconstruction process are complex and lie beyond the scope of this paper. For a simple example, we include our variant for the lazy *pdg* for the following simple program:

$$
\begin{aligned}
R \stackrel{df}{=}\ & s := 0\,;\, x := 1\,; \\
& \textbf{while } x \leq N \textbf{ do} \\
& \quad s := s + x\,; \\
& \quad x := x + 1
\end{aligned}
$$

The result appears in Figure 9. Edges with a bar represent loop-carried data dependence edges, others are either control or data dependences.

5 Directions for Future Research

Although the semantic analysis presented in this paper is instructive, it is too narrow to serve as a practical framework for expressing and justifying program optimizations. To satisfy this goal, we must expand the programming language to include the all of the fundamental operations included in real programming languages, namely composite data structures (arrays, pointers and records), procedures, and more general control structures. In conjunction with this effort, we also intend to develop an equational calculus for reasoning about *pdgs*. Given this machinery, it should be possible to construct an optimization laboratory that enables compiler writers to specify optimizing transformations in concise, formal notation; to generate experimental optimizing compilers from these specifications; and to prove the correctness of the incorporated optimizations.

Acknowledgements. We gratefully acknowledge several discussions with Joe Warren, Thomas Reps, and Susan Horwitz.

References

1. Aho, A., R. Sethi, and J. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1985.
2. Ferrante, J. and K. Ottenstein. A program form based on data dependency in predicate regions. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983, 217–236.
3. Ferrante, J., K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9(3), 1987, 319–349.
4. Horwitz, S., J. Prins, and T. Reps. On the adequacy of program dependence graphs representing programs. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, 146–157.
5. Horwitz, S., J. Prins, and T. Reps. Integrating non-interfering versions of programs. In

Proc. 15th ACM Symposium on Principles of Programming Languages, 1988, 133–145.

6. JØRRING, U. AND W. L. SCHERLIS. Compilers and staging transformations. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, 86–96.
7. KUCK, D. J., R. H. KUHN, D. A. PADUA, AND M. WOLFE. Dependence graphs and compiler optimizations. In *Proc. 8th ACM Symposium on Principles of Programming Languages*, 1981, 207–218.
8. OTTENSTEIN, K. J. An intermediate program form based on a cyclic data-dependence graph. Technical Report No 81-1, Department of Computer Science, Michigan Tech. University, 1981.
9. SCHMIDT, D.A. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Newton, Mass., 1986.
10. SELKE, R. P. A Rewriting Semantics for Program Dependence Graphs. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989, 12–24
11. WAND, M. Loops in combinator-based compilers. In *Proc. 10th Symposium on Principles of Programming Languages*, 1983, 190–196.
12. WAND, M. Semantics-directed machine architecture. In *Proc. 9th Symposium on Principles of Programming Languages*, 1982, 234–241.
13. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* 4(3), 1982, 496–517.
14. WARREN, J. A hierarchical basis for reordering transformations. In *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984, 272–282.
15. WOLFE, M. J. Optimizing Supercompilers for Supercomputers. Ph. D. dissertation, University of Illinois, 1982.

Appendix

The definitions of the semantic functions $\mathcal{G}$ and $\mathcal{P}$ rely on the domain constructors $\otimes$ (product), $\oplus$ (sum), $^+$ (infinite sequences), and $\multimap$ (finite functions) together with the associated functions $\langle\cdot,\cdot\rangle$ and $\circ$ for constructing pairs and infinite sequences, respectively. The domain operators $\otimes$, $\oplus$, and $^+$ are defined by the following equations:

$$\begin{aligned} A \otimes B &= \{\bot\} \cup \\ &\quad \{(a,b) \mid a \in A \setminus \{\bot\} \wedge b \in B \setminus \{\bot\}\} \\ A \oplus B &= \{(\mathbf{T},a) \mid a \in A \setminus \{\bot\}\} \cup \\ &\quad \{(\mathbf{F},b) \mid b \in B \setminus \{\bot\}\} \cup \{\bot\} \\ A_\bot &= \{\bot\} \cup \{(\mathbf{T},a) \mid a \in A\} \\ A^+ &= A \otimes (A^+)_\bot \end{aligned}$$

The values $\mathbf{T}$ and $\mathbf{F}$ are used as "tags" in the construction of composite objects; the objects $(\mathbf{T},a)$ and $(\mathbf{F},a)$ are distinct from $\bot$ regardless of the value of a (including $a = \bot$). The binary function

$$\langle\cdot,\cdot\rangle : A \times B \longrightarrow A \otimes B$$

constructs elements of $A \otimes B$ as follows:

$$\langle a,b\rangle = \begin{cases} (a,b) & \text{if } a \neq \bot \wedge b \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

The expression $\langle e_1, e_2, \ldots, e_n\rangle$ abbreviates

$$\langle e_1, \langle e2, \ldots\rangle\rangle$$

Similarly, the expression $[e_1, e_2, e_3, \ldots]$ abbreviates

$$\langle e_1, (\mathbf{T}, \langle e_2, (\mathbf{T}, \langle e_3, (\mathbf{T}, \ldots)\rangle)\rangle)\rangle$$

The infix operator

$$\circ : A \times A^+ \longrightarrow A^+$$

is defined by:

$$\begin{aligned} a_0 \circ [a_1, a_2, \ldots] &= \langle a_0, \langle \mathbf{T}, [a_1, a_2, \ldots]\rangle\rangle \\ &= [a_0, a_1, a_2, \ldots]. \end{aligned}$$

The domain $A \multimap B$ is the set of all *finite* subsets S of $A \otimes B$ that correspond to functions: $(a,b) \in S$ and $(a,b') \in S$ implies $b = b'$. We frequently interpret elements of $A \multimap B$ as functions in $A \longrightarrow B$.