Dependence Flow Graphs:
An Algebraic Approach to Program Dependencies*

Keshav Pingali
Micah Beck
Richard Johnson
Mayan Moudgill
Paul Stodghill

TR 90-1152
September 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported by an NSF Presidential Young Investigator award (NSF Grant No. CCR-895854), and by grants from the Digital Equipment Corporation and IBM.

Dependence Flow Graphs: An Algebraic Approach to Program Dependencies

Keshav Pingali
Micah Beck Richard Johnson
Mayan Moudgill Paul Stodghill

Department of Computer Science
Cornell University
Ithaca, NY 14853.

Abstract

The topic of intermediate languages for optimizing and parallelizing compilers has received much attention lately. In this paper, we argue that any good representation must have two crucial properties: first, the representation of a program must be a data structure that can be rapidly traversed to determine dependence information; second, the representation must be a program in its own right, with a parallel, local model of execution. In this paper, we illustrate the importance of these points by examining algorithms for a standard optimization — global constant propagation. We discuss the problems in working with current representations. Then, we propose a novel representation called the *dependence flow graph* which has each of the properties mentioned above. In this representation, dependencies are part of the computational model, in that there is an algebra of operators over dependencies. We show that this representation leads to a simple algorithm, based on abstract interpretation, for solving the constant propagation problem. Our algorithm is simpler than, and as fast as, the best known algorithms for this problem. An interesting feature of our representation is that it naturally incorporates the best aspects of many other representations, including continuation-passing style, data and program dependence graphs, static single assignment form and dataflow program graphs.

[1]This research was supported by an NSF Presidential Young Investigator award (NSF grant #CCR-8958543), and by grants from the Digital Equipment Corporation and IBM.

[2]Comments regarding this paper should be directed to pingali@cs.cornell.edu.

1 Introduction

The growing complexity of optimizing and parallelizing compilers has re-focused the attention of the programming languages community on the design of *intermediate program representations*. Some well-known representations are: control flow graphs [ASU86], def-use chains [ASU86], data dependence graphs [Kuc78], program dependence graphs and webs [FOW87, BMO90], program representation graphs [CF89], static single assignment form [CFR+89], continuation-passing style [SS78] and program graphs [Ack84]. The choice of program representation has a profound effect on the design, asymptotic complexity, and implementation of optimizing and parallelizing transformations. As an analogy, consider Hindu numerals[3], which are more convenient than Roman numerals for performing arithmetic operations, while representing the same information. In this paper, we argue that a good intermediate representation should have the following properties:

- It should be executable. That is, it should be a language with a well-defined, compositional operational semantics. This allows abstract interpretation to be employed when designing algorithms, which facilitates systematic algorithm development and proof of of correctness [CC77, CC79].

- It should be possible to view the representation as a data structure that can be efficiently traversed for data dependence information, as needed used many compiler transformations [Kuc78].

- Loops should be represented explicitly. Some representations replace loops with tail-recursive procedures in [AA89, Eka90]. In our experience, this transformation is not desirable since many important loop transformations, such as loop interchange, have no natural analog in the context of tail-recursive procedures.

- The storage model should include an updatable, imperative store. The operational semantics of an imperative language can be phrased naturally in terms of an updatable store. While it is possible to treat the store functionally (as is done in denotational semantics), such treatments are quite clumsy in dealing with data structures, especially arrays [ANP89].

- The representation should be compact. A new program representation whose size is asymptotically bigger than that of well-accepted representations (such as def-use chains) is unlikely to gain acceptance.

In this paper, we illustrate the importance of these issues by examining a particular optimization — global constant propagation. This optimization is performed by all optimizing compilers and is representative of "scalar" optimizations such as partial redundancy elimination and strength reduction. We discuss the drawbacks of the representations cited

[3]Because of an unfortunate instance of aliasing, these are known as Arabic numerals in the West.

above, and demonstrate how a representation that meets our criteria leads to a simple, elegant algorithm, based on abstract interpretation. This algorithm is as efficient as the best algorithms that use the other forms, and has an elegant proof of correctness. Our representation, called the *dependence flow graph*, is based on a generalization of the dataflow model of computation, called *dependence-driven execution.* Interestingly enough, many features of previously proposed intermediate representations arise naturally in the context of dependence flow graphs.

This paper makes the following contributions:

- For compiler writers, we propose a novel intermediate program representation that has demonstrable advantages over existing representations. This intermediate representation can be used for both functional and imperative languages.

- For designers of optimization algorithms, we demonstrate how the simple and powerful technique of abstract interpretation can be used without loss of efficiency on dependence flow graphs. At present, this technique is not widely used because abstract interpretation on control flow graphs is inefficient.

- We describe a novel algorithm for global constant propagation. This algorithm is simpler than, and as efficient as, the most powerful algorithms to date, such as the one due to Wegman and Zadeck [WZ84].

- For dataflow researchers, we demonstrate a way of implementing imperative languages like FORTRAN on dataflow machines. To date, these machines execute only functional languages. The availability of FORTRAN will make these machines acceptable to a much wider group of users.

- We propose a provocative view of the future of the dataflow model of computation [Den74]. Conventionally, the dataflow model is viewed as a way of organizing parallel architectures for executing functional language programs, but these ideas have not had a major impact on mainline architectures. Our results suggest that the dataflow model of computation may yet find its destiny as a way of organizing information — in imperative language (FORTRAN) compilers!

The rest of this paper is organized as follows. In Section 2, we illustrate the drawbacks of previously proposed representations by describing how constant propagation is performed using them. In Section 3, we present dependence flow graphs and give a formal, Plotkin-style operational semantics for these graphs. In Section 4, we show a simple algorithm, based on abstract interpretation, that uses dependence flow graphs to solve the global constant propagation problem, and we prove the algorithm correct. Finally, in Section 5 we place our results in perspective and discuss ongoing work.

2 Constant Propagation

In this section, we examine the problem of global constant propagation, a standard analysis performed by optimizing compilers. In Section 2.1 we define a particularly ambitious class of constants, the *possible-paths constants*, which is discovered by an algorithm due to Wegman and Zadeck [WZ84]. We then consider a number of intermediate forms most commonly used for optimization in imperative language compilers: the control flow graph in Section 2.2, the data dependence graph in Section 2.3, and the control flow graph with def-use chains in Section 2.4. We show how constant propagation algorithms that use these representations are affected by the shortcomings of the underlying representations. Finally, we discuss the lessons learned from this comparative study.

2.1 Problem Description

A *definition* of a variable x is a statement that assigns (or may assign) to x. A *use* of x is an occurrence of x in a statement that reads (or may read) the value of x. We say that a definition of x *reaches* a use of x if execution of the definition may be followed by execution of the use without intervening execution of any other definition of x. As is standard, this definition assumes that conditional branches may go either way [ASU86].

If the right hand side of a definition of x is a constant c, we can sometimes substitute c for a use of x without changing the meaning of the program. For example, in Figure 1(a), the first use of z can be replaced by 1 and the second by 2. This is a simple example of *constant propagation.* If all of the variables on the right hand side of a definition are replaced by constants, then we can evaluate the expression and replace it by a constant. Recursively, this opens up fresh opportunities for constant propagation. For example, in Figure 1(a), the right hand sides of the two definitions of x can be simplified to the constant 3. The use of x in the last statement is reached by *two* definitions of x. However, since the right hand sides of both definitions are the same constant, we can replace the use of x by 3 without changing the meaning of the program. This motivates the following definition.

An *all-paths constant* is either:

- a constant expression c, or
- an expression e over some set of variables $\{v_1, v_2, \ldots v_n\}$ such that for each v_i, the right hand side of every definition of v_i that reaches e is an all-paths constant c_i.

The class of all-paths constants takes no account of constants in conditionals. However, if the predicate of a conditional can be determined to be constant, then we can ignore the effect of definitions on the side that is never executed. If we modify the definition of all-paths constants to exclude such definitions, the result is the class of *possible-paths constants* [WZ84]. In Figures 1(b) and (c), the use of x in the last statement is a possible-paths constant with value 1. Note that neither of these uses is an all-paths constant.

A variety of algorithms for constant propagation have been proposed in the literature [ASU86, Kil73, RL77, WZ84]. Some of these algorithms are more powerful than others

```
if (p) then { z := 1; x := z+2 }
        else { z := 2; x := z+1 }
y := x
```

(a) all-paths

```
p := true
if (p) then { x := 1 }
        else { x := 2 }
y := x
```

(b) possible-paths

```
x := 2
p := true
if (p) then { x := 1 }
y := x
```

(c) one-sided possible-paths

Figure 1: Examples of runtime constants

— for example, only the algorithm of Wegman and Zadeck [WZ84] finds possible-paths constants in a single pass. Note that repeated application of the less powerful algorithms, combined with dead code elimination, will in fact find the possible-paths constants. Since repeated rounds of program transformation and analysis are expensive, we seek to discover as many constants as possible in a single pass through the program. As we will see, the choice of program representation plays a critical role in this task.

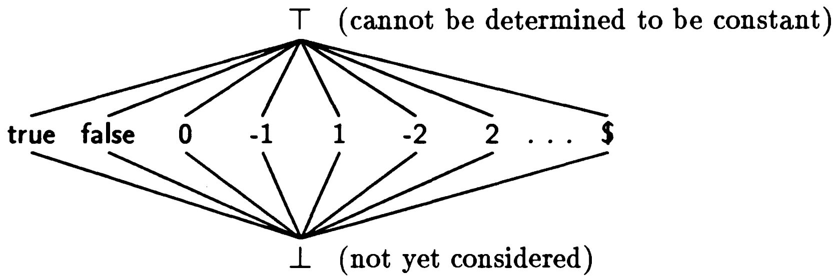


Figure 2: *Lat* — Lattice for constant propagation

It is standard to express constant propagation algorithms in the framework due to Kildall [Kil73]. We define a lattice *Lat* shown in Figure 2, consisting of all the constant values and two distinguished values $\top$ and $\perp$. The special constant **$** is used only in dependence flow graphs, and plays no part in the algorithms described in this section. Uses of variables are assigned values from *Lat* during constant propagation. Initially, every use of every variable is mapped to $\perp$, meaning that we have no information yet about the values that it is assigned at runtime. A use is mapped to $\top$ when the algorithm cannot determine that the use is a constant (*e.g.* if the use is reached by two definitions whose right hand sides are 3 and 4.)[4] At the end of constant propagation, the interpretation of the lattice value assigned

[4]Note that the sense of $\top$ and $\perp$ in the lattice are reversed with respect to the lattice used by previous researchers [Kil73, RL77, WZ84]. These researchers viewed constant propagation as an all-paths data flow problem; such problems are traditionally formulated so that the desired solution is the *greatest* fixed point of a set of equations. In our framework, we will use abstract interpretation to find constants, and it is more convenient to formulate the desired solution as the *least* fixed point of a set of equations.

to a use of a variable x is as follows:

- $\perp$ The use was never examined during the constant propagation algorithm; it is dead code.
- c At this point, x will always have the value c.
- $\top$ At this point, x may have different values in different executions.

To permit evaluation of right hand sides of definitions, it is convenient to extend the usual arithmetic and boolean operators so that they can take $\perp$ and $\top$ as arguments. For example, the operator v1 + v2 is interpreted as follows:

$$Plus(v_1, v_2) = \begin{cases} \top & \text{if } v_1 = \top \text{ or } v_2 = \top \\ c_1 + c_2 & \text{if } v_1 = c_1 \text{ and } v_2 = c_2 \\ \perp & \text{otherwise} \end{cases}$$

2.2 Control Flow Graphs

Figure 3(b) shows the *control flow graph* [ASU86] for a small imperative program. Nodes are either assignment statements or conditional expressions that affect flow of control, and edges represent possible transfer of control between nodes. An assignment node has a single successor while a conditional node has two successors representing the possible branching of control. In our figures, we follow the convention that the *true* branch of a conditional is always left-most. Algorithms for constructing the control flow graph representation of a program are well-known [ASU86].

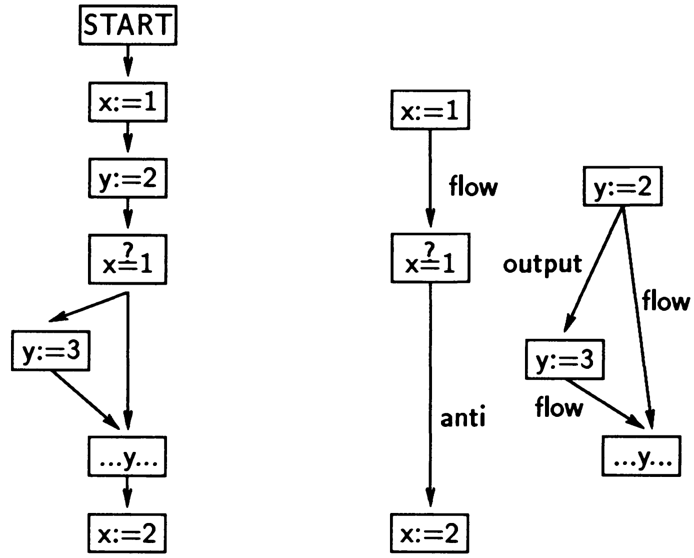
Control flow graphs are an executable representation, and we give an informal semantic model. The model we describe is of the "token pushing" style generally used to describe execution on dataflow graphs [Den74], in order to set the stage for our presentation of the semantics of dependence flow graphs in Section 3. A single "control token" flows through the graph and its arrival at a node signifies that the node may execute.[5] To begin execution, the control token is emitted by the distinguished node labeled START. A global store is used to hold the values of all variables. When a node executes, it accesses this global store, reading from locations, performing computations and writing into the store as necessary. After an assignment node executes, it emits the control token along its single outedge and enables execution of the destination node. When a conditional expression executes, the control token flows along the outedge representing the selected branch and enables execution of the destination node. In both cases execution of a node enables a single successor to execute; hence the execution model is completely sequential.

2.2.1 Constant Propagation on the Control Flow Graph

We describe an algorithm based on abstract interpretation that finds possible-paths constants in the control flow graph. At each node, we maintain a vector of values from *Lat*. These

[5] The control token plays the same role as a program counter in the von Neumann model of execution.

```
x := 1
y := 2
if  (x ≟ 1)
    then y := 3
...y...
x := 2
```



(a) Source Program (b) Control flow Graph (c) Data Dependence Graph

Figure 3: A Small Program and its Representations

vectors have an entry for each variable, and intuitively, they summarize the possible values of variables before the statement is executed. Initially, every entry at every node is set to $\perp$ except at the START node where the entries are set to $\top$. These values are updated monotonically (in the lattice-theoretic sense) as the algorithm proceeds.

The algorithm maintains a worklist of nodes to be processed; initially, this worklist contains all of the nodes immediately following the START node. Nodes are dequeued from the worklist and processed as follows. Let N be a node from the worklist, and let N_{in} be the vector at the input of N.

- If the node is an assignment statement, (say x := **e**), then the expression **e** is evaluated, using the values of variables in N_{in}, and a new vector N_{out} is created that is identical to N_{in} except at x where it has the value just computed for **e**. The vector N_{out} must be propagated to the successor node S of the assignment statement. Let S_{in} be the vector at the successor node. This vector is updated to the join of its value and N_{out}.[6] If this changes the value of S_{in}, then S is added to the worklist.

- If the node is a conditional branch, then the predicate is evaluated. If the value of the predicate is $\top$, then the vector N_{in} is propagated to both successors of the conditional branch. If the value of the predicate is *true* or *false*, then N_{in} is propagated only along the corresponding side of the conditional branch. If the value of the input vector changes at the successor(s), then they are added to the worklist.

[6]We cannot just store N_{out} at S_{in} because S may have other predecessors. If a node has two predecessors, $x := 2$ and $x := 3$, then the entry for x at its input should be $\top$.

We leave it to the reader to verify that this algorithm will find all of the constants in Figure 1. Unfortunately, the asymptotic complexity of this algorithm is poor. Let V be the number of program variables and N the number of statements. Then V values must be stored at every statement, so the space requirement of the program is $O(NV)$. Every variable can be promoted in the lattice at most twice at every statement, so each statement can be symbolically executed at most $2V$ times. Each execution of a statement requires $O(V)$ operations, so the total complexity of the algorithm is $O(NV^2)$.

2.2.2 Discussion

Although the abstract interpretation algorithm on control flow graphs is simple, it is not used in practice because of its high cost. The inefficiency arises because lattice values must be propagated along control flow paths from definitions of variables to their uses. If there are n statements between a definition and a use of a variable, then the lattice value produced at the definition must be propagated through these n statements, even if they do not reference the variable! What is needed is a "sparse" representation that links definitions to the uses they reach, so that we can propagate the values of individual variables to places where they are needed, rather than propagating the values of all variables to all program locations. Def-use chains and their generalization, data dependence graphs, provide such a representation.

2.3 Data Dependence Graphs

Def-use chains are graphs that have the same nodes as control flow graphs, but the arcs connect each definition of a variable to all uses reached by that definition. For compilers that perform wholesale reorganization of programs, a generalization of def-use chains called the *data dependence graph* [Kuc78] is commonly used. The data dependence graph for our example is shown in Figure 3(c). Arcs in the graph represent dependencies that are classified as follows:

- *Flow dependencies*: These are def-use chains; the source defines x and the destination is a use of x reached by the definition.
- *Anti dependencies*: The source uses x and subsequently the destination reassigns x.
- *Output dependence*: The source assigns to x and subsequently the destination reassigns x.

Data dependencies can be interpreted as execution ordering constraints between nodes: if both the source and destination are executed, then the source must be executed before the destination. In programs with loops, one must be careful to distinguish between the static arc in the dependence graph and the execution order implied between dynamic instances of those nodes. Within a loop, a dependence arc may order two operations in the same dynamic iteration (a *loop independent* dependence), or it may order operations in successive iterations (a *loop carried* dependence). For example, a statement such as x := x + 1 inside a loop may have a loop carried flow dependence on itself since it may assign to x and may then read the value in the next iteration.

The data dependence graph is not an executable representation and does not incorporate information about flow of control. For example, in Figure 3(c), execution of the definition y := 3 is not related to the predicate $x \stackrel{?}{=} 1$ in any way. Negating the predicate will change the value of y that is read, but this does not change the dependence arcs that sequence operations on y.

2.3.1 Constant Propagation on the Data Dependence Graph

The constant propagation algorithm based on def-use chains is similar to the one described earlier for control flow graphs, except that we propagate lattice values from definitions to uses along def-use arcs, rather than along control flow paths. At each node, we keep a vector containing values only for those variables that are used by the node. A worklist is kept of nodes to be processed. Initially, every definition whose right hand side is a constant c is placed on this worklist.

To process a definition of the form x := **e**, the expression **e** is evaluated as described in Section 2.1, and its value is propagated along def-use links originating at this node to nodes that use x. The update is performed by setting the entry for x in the input vector of the node to the join of the value of **e** and the entry for x in the input vector. If this changes the input vector at the destination node of the def-use link, and the node is a definition, then it is added to the worklist. Conditional nodes do not need any processing, since there are no def-use chains originating at these nodes.

The complexity of this algorithm is linear in the size of the def-use chains of the program. A naive representation of def-use chains can be $O(E^2V)$ in size, where E is the number of arcs in the control flow graph (E can be at most $2N$). Reif and Lewis have shown that a factored form of def-use chains can be represented in size $O(EV)$. This complexity is better than that of the algorithm that uses control flow graphs. Unfortunately, although the algorithm will find all-paths constants as in Figure 1, it will not find possible-paths constants such as those in Figure 1(b) or (c). The problem is that, in relying on data dependence information to direct the flow of values, we lose important connections between data dependence and flow of control. In our control flow algorithm, values were propagated down only one side of a conditional with a constant predicate. This was possible because the control flow path from the definition to the use passed through the conditional. Flow dependence edges do not flow through conditionals, but reach directly from definitions to uses. Thus, our new algorithm does not catch possible-paths constants.

2.3.2 Discussion

We conclude that def-use information needs to be augmented with control flow information in some way. Two alternatives have been studied in the literature. The first alternative is to use both the control flow graph and the data dependence graph in performing program optimization. The second alternative is to add a new kind of dependence arc called *control* *dependence* to the data dependence graph. The resulting data structure is called the program dependence graph.

2.4 Control Flow Graphs with Data Dependence Graphs

Most optimizing compilers generate a control flow graph as a first step towards computing the data dependence graph. Since both representations are available, it is possible to design "hybrid" algorithms that use both the control flow graph and the data dependence graph. The constant propagation algorithm described next is adapted from that of Wegman and Zadeck [WZ84].

2.4.1 Constant Propagation on a Hybrid Representation

To find possible-paths constants while still obtaining the efficiency of def-use chains, Wegman and Zadeck refer back to the control flow graph. To keep propagation of values from bypassing conditionals, a boolean *executable* flag is added to each statement, and is initially set to *false*, except for START. The executable flag indicates that the statement may be executed, *i.e.,*that it has not been determined to be dead.

Lattice values flow along def-use chains as before; in addition, information about which nodes may execute flows through the control flow graph. These two flows are not independent since intermediate results of constant propagation may be used to determine that one side of a conditional is never executed. Conversely, a definition does not participate in constant propagation until it is determined that it will never be identified as dead. Two worklists keep track of these two flows: the *flow* worklist and the *def* worklist. The flow worklist is used to propagate information, through the control flow graph, about which nodes are executable. The *def* worklist, is used to propagate lattice values along def-use arcs. Initially, the flow worklist contains only START and the def worklist is empty.

Execution proceeds while either worklist is nonempty; a node is taken off of either the flow or def worklist, and processed as follows:

1. Suppose the node is taken off the flow worklist. If the node is START, or is a definition, then the executable flag of the successor node is set to *true*. Otherwise, the node is a conditional, so the conditional expression is evaluated. If the predicate is $\perp$, then no action is taken[7]. If the predicate is *true* or *false*, then the executable flag of the control flow successor on the corresponding side of the branch is set to *true* — if the predicate is $\top$, then the executable flag of the control flow successors on both sides of the branch are set to *true*.

   If the executable flag on a successor node is changed from *false*, to *true*, then these nodes are added to the flow worklist; definition nodes are also added to the def worklist.

[7]This means that lattice value propagation has not yet reached this node, so the node will be re-examined later.

```
x := 2
p := true
if (p) then { x := 1 }
y := x
```

(a) one-sided possible-paths

```
x := 2
p := true
if (p) then { x := 1 }
     else { x := x }
y := x
```

(b) with dummy assignment

Figure 4: Transforming a one-sided conditional

2. If the node is taken off the def worklist, then execution of the node proceeds as in the previous algorithm, except that only nodes with executable flag set to *true* are added to the worklists; an executable successor node that is a conditional is added to the flow worklist, and all other executable successor nodes are added to the def worklist.

This algorithm inhibits the flow of lattice values from definitions that can never be executed. It will find the constants in Figure 1(a) and (b), but fails to discover the one-sided possible-paths constant in Figure 1(c), since the assignment x := 2 reaches the use of x and is not dead. Wegman and Zadeck suggest transforming every one-sided conditional by inserting a dummy assignment of the form x := x on the **else** branch. In the transformed program, as in Figure 4(b), only one value is propagated through the conditional if the predicate is constant. When performed on the transformed program, the Wegman-Zadeck algorithm does find the possible-paths constants.

2.4.2 Discussion

It is more difficult to use the hybrid representation consisting of the control flow graph and the data dependence graph than a single representation, as is clear by comparing the constant propagation algorithms. When performing program transformations, it is difficult to keep the two representations consistent. In fact, many compilers perform transformations only on the control flow graph, and update the data dependence graph periodically by redoing flow analysis on the transformed control flow graph [ASU86]. This is expensive, and also means that transformations cannot take advantage of previous transformations since the data dependence information is not up-to-date.

The problem of maintaining two data structures to represent the program's execution semantics and its dependencies is addressed in part by the *program dependence graph.* This graph consists of the data dependence graph augmented with *control dependence* arcs. In the case of constant propagation a more elegant algorithm based on the one described in the preceding section can be developed using the program dependence graph. However, program dependence graphs inherit many of the problems of the data dependence graph; for example, for constant propagation, we must still perform the program transformation shown in Figure 4. Moreover, they do not have a simple, local execution semantics [CF89]

2.5 Summary

The control flow graph allows us to formulate a simple algorithm, based on abstract interpretation, that found possible-paths constants without the need for program transformations. However, its asymptotic complexity was poor. Algorithms that used the various dependence graphs were more complex, and none of them could find possible-paths constants without some program transformation. However, the asymptotic complexity of these algorithms was a factor of V better than the algorithm that used control flow graphs.

The difficulty with the various dependence graphs is that control information, if present at all, is not combined in any way with data dependence information. This makes it difficult to determine which data dependencies should be eliminated from consideration once some predicate is determined to be constant. This is an example of a more general problem: it is difficult to update dependence graphs to reflect the result of program transformations that result in the addition or deletion of nodes. By contrast, notice that it is trivial to insert or delete instructions from control flow graphs. The fundamental difference is that control flow graphs are an *executable* representation with a *local* semantics — if we imagine that a node is an active, autonomous entity with certain inputs, it can monitor those inputs to determine when it should execute, and what the result of its execution should be. This is difficult to do with dependence graphs, and this difficulty is directly related to the difficulty of updating these graphs incrementally to reflect the results of program transformation.

We conclude that an ideal representation should permit two points of view, like a Necker cube — it can be viewed as a data structure that can be traversed efficiently for dependence information, but it can also be viewed as a precisely defined language with a local operational semantics. We now describe such a representation.

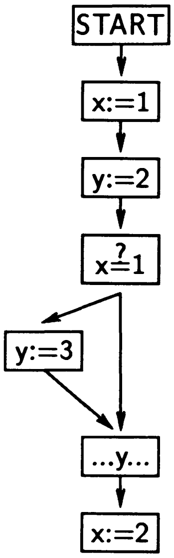
3 Dependence Flow Graphs

Figure 5(d) shows the dependence flow graph for the imperative language program considered in Section 2. Dependence flow graphs can be viewed as a synthesis of ideas from data dependence graphs and the dataflow model of computation. As in the data dependence graph, the dependence flow graph can be viewed as a data structure in which arcs represent dependencies between operations. In Figure 5, it is easy to verify that for every dependence arc in the data dependence graph, there is a corresponding path in the dependence flow graph. However, unlike data dependence graphs, dependence flow graphs are *executable*, and the execution semantics, called *dependence-driven execution*, is a generalization of the data-driven execution semantics of dataflow graphs. In dataflow graphs, nodes represent functional operators that communicate with each other by exchanging value-carrying tokens along arcs in the graph. These arcs can be viewed as flow dependencies since they connect a node producing a value, such as an integer or boolean, to nodes that consume this value; we will call such arcs *functional dependencies* in our presentation. In Figure 5(d), `v`, `v1` and `b` are functional dependencies.

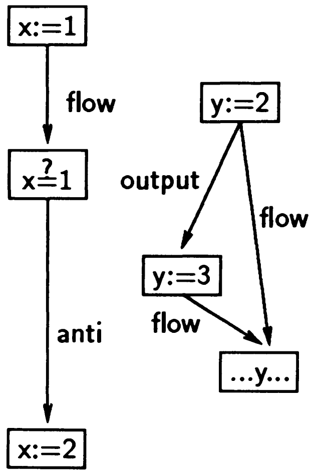
We extend the dataflow model by adding an imperative (updatable) global store and

```
x := 1
y := 2
if  (x =? 1)
    then { y := 3 }
...y...
x := 2
```

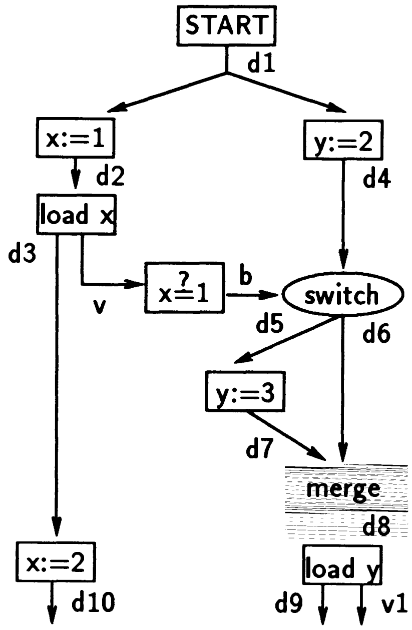
(a) Source Program



(b) Control flow Graph



(c) Data Dependence Graph



v, v1	: integer
b	: boolean
d1, d4, d5	: output
d2, d4, d6, d7, d8, d9, d10	: flow
d3	: anti

d1	= START()
d2	= store(x,1,d1)
v,d3	= load(x,d2)
b	= equal(v, 1)
d4	= store(y,2,d1)
d5,d6	= switch(b, d4)
d7	= store(y3,d5)
d8	= merge(d7,d6)
v1,d9	= load(y,d9)
d10	= store(x, 2)

(d) Dependence Flow Graph

Figure 5: A Small Program and its Representations

two operations called load and store which manipulate it. As one would expect, the load operator reads the contents of a storage location and outputs the value as a token. The store operator is the inverse of the load operator — it receives a value on a token and stores it into a memory location. To sequence these operations, we introduce a new kind of arc called an *imperative dependence*. For example, in Figure 5(d), d2 and d3 are imperative dependencies that sequence operations on location x, corresponding to arcs in the data dependence graph. To preserve the local, token-pushing semantics of dataflow graphs, we make load and store operators produce a special token, $, when they have completed. These tokens flow down imperative dependence arcs to enable operators at the destinations of those arcs. For example, when the x := 1 operator executes, it produces a token carrying $ on line d3. This is said to *satisfy* the dependence d3, thereby *enabling* the load x operator for execution. When the load x operator executes, it produces tokens carrying $ on line d3 and the value 1 on line v. In this way, operations on a given memory location are sequenced, but operations on different locations can execute in parallel.

Imperative dependencies are further classified as flow, anti and output as in data dependence graphs. We classify d2 as a flow dependence and d3 as an anti-dependence. Note that dependence d5 is both a flow and an output dependence, since logically it corresponds to both of the dependence arcs coming out of the definition y := 2 in the data dependence graph. Dependence arcs that sequence operations on location y are intercepted by switch and merge operators, which implement flow of control as discussed below. These operators serve to combine control information with data dependencies, which is exactly what is missing in the representations discussed in Section 2.

To understand dependence flow graphs, it is useful to execute the graph depicted in Figure 5(d) by pushing tokens. Execution begins when the START operator sends a token carrying $ to the store operations x := 1 and y := 2. Depending on whether the token received on arc b is true or false, the switch operator outputs the token it receives on d4 onto either arc d5 or d6. In our example, the switch routes the token to d5, and the definition y := 3 is executed. The merge operator receives a token on either one (but not both) of its inputs, and simply outputs this token. The reader can verify that a token carrying the value 3 will be generated on arc v1.

In a forthcoming paper, we will describe how dependence flow graphs are constructed, starting from the control-flow graph of a program. This construction can handle unstructured control-flow. Some preliminary ideas are presented in an earlier paper [BP90]. From an analysis of the construction, we show two facts.

- Dependence flow graphs constructed by our algorithm satisfy Bernstein's conditions: that is, a store operator can never be enabled for execution simultaneously with another store or load operator on the same storage location [Ber66].

- The dependence flow graph of a program whose control flow graph has E edges and V variables has size $O(EV)$.

Although token-pushing provides useful intuition, we adopt a different style of operational semantics in the formal development. Arcs in the dependence flow graph can be viewed as names that represent a set of single-assignment registers or temporaries. Producing a token carrying a value on an arc is similar to storing that value in the corresponding register. Explicit load and store operators to transfer values between the global store and a set of registers/temporaries have been used in the PL.8 compiler [AH82] and many Scheme compilers [SS78]. We develop this point of view in the rest of this section.

3.1 Acyclic Dependence Flow Graphs: Formal Semantics

From a formal perspective, a dependence flow graph is a set of *declarations* followed by a set of *definitions*. Declarations introduce names for locations in the store and for *dependencies*, which can be viewed as names for a set of single-assignment temporaries or registers. The body of the dependence flow graph is a set of definitions. A definition is an equation with a left hand side consisting of one or more dependencies, and a right hand side consisting of the application of an operator to dependencies, locations and constants. The operators and their arity are shown in the left column of Figure 6. A definition is said to be a *source* for dependencies named on the left hand side of the equation and a *sink* for dependencies named on the right hand side. A dependence has exactly one source but can have many sinks.

We now give a Plotkin-style, formal operational semantics for dependence flow graphs [Plo81]. Rather than rewrite programs, as is common in this style of semantics, we will define a state transition semantics in which we rewrite *configurations*. Informally, a configuration represents the state of the computation and a transition represents a step in the computation. In our system, a configuration is a pair consisting of an *environment* and a *store*. To define them, we need the following sets.

- $V = \mathit{Bool} \cup \mathit{Int} \cup \{\$\}$ is the set over which we compute. The metavariables b and c stand for elements of V.
- $\mathrm{Loc} = \{L_0, L_1, ...\}$ is an infinite set of global store locations. The metavariable x stands for an element of Loc.
- $\mathrm{Dep} = \{d_0, d_1, ...\}$ is an infinite set of dependencies. The metavariables d, v, t and b stand for elements of Dep.

The environment keeps track of the state of dependencies in the program and the store keeps track of the state of locations used by the program. The environment is a mapping from program dependencies to the set V. For technical reasons, a dependence will be added to the environment only when it is satisfied; therefore, the initial environment is empty. The environment grows monotonically during execution, in the sense that as computation progresses, dependencies are only added to and never deleted from the environment; in addition, the value bound to a functional dependence in the environment never changes. Similarly, the store is a mapping from the set of locations used in the program to the set V; however, locations can be updated arbitrarily. The store, like the environment, is initially empty and a location is added to the store the first time a value is stored to it.

Definition 1

1. *An* environment $\rho : D \rightarrow V$ *is a finite function — its domain* $D \subset Dep$ *is finite.*

2. *A dependence* d *is said to be* defined *or* satisfied *in* ρ *if* d *is in the domain of* ρ. *Otherwise, it is said to be* undefined *in* ρ. *The notation* $\rho[v \mapsto c]$ *represents an environment identical to* ρ *except for dependence* v *which is mapped to* c.

3. *A* store $\sigma : L \rightarrow V$ *is a finite function — its domain* $L \subset Loc$ *is finite. Just as for dependencies, we can talk about a location* x *being defined or undefined in a store* σ.

4. *A* configuration *is a pair* $\langle \rho, \ \sigma \rangle$ *consisting of an environment and a store. The initial configuration has empty environment and store.*

Figure 6 shows the transition rules for acyclic dependence flow graphs. The left column consists of definitions and the right column shows a precondition on top of the line, and a transition below the line. If the definition in the left column is present in the dependence flow graph and the precondition on top of the line is satisfied, then the transition shown below the line can be performed.

As an example, consider a definition of the form $t = \textsf{add}\ (t_1, t_2)$. We would expect this operator to execute when t_1 and t_2 are defined. Once this operator has executed, we want to disable this transition. Therefore, we perform the transition only if t is undefined. The **load** and **store** operators are the only operators that access the store. The **load** operator checks that the contents of location x are defined; this will catch an attempt to read from an uninitialized location. The **switch** and **merge** operators implement flow of control. Depending on the boolean value b, the switch operator satisfies either dependency t_{true} or t_{false}. The dependence at the output of the **merge** is satisfied when either of the dependencies at its input is satisfied. Notice that the rules check that at most one input dependence is satisfied.

3.2 Cyclic dependence flow graphs

As far as the input/output behavior of programs goes, loops can be replaced by tail-recursion. However, many loop optimizations, such as loop interchange, have no natural analog in the context of tail-recursion, so we felt it was important to model loops directly using cyclic dependence flow graphs. For this, we need two new operators called **loop** and **until** which are used at loop entrance and loop exit respectively. In addition, the transition rules for the operators discussed in Section 3.1 must be altered slightly.

Consider the definition $t = \textsf{add}\ (t_1, t_2)$ occurring inside a loop. t represents a different dependence in each iteration; to model this we index t by the iteration number, so that $t.i$ represents the dependence t in the i^{th} loop iteration.[8] $t.1$ is the dependence in the first iteration. This scheme extends naturally for nested loops so that for a two-dimensional loop, $t.i.j$ represents this dependence in iteration i of the outer loop and iteration j of the inner

[8] This device is like scalar expansion [PW86]

Transition Rules for Acyclic Dependence Flow Graphs

$d = \textsf{start}\ (\,)$:
$$\frac{d \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[d \mapsto \$],\ \sigma\rangle}$$

$t = \textsf{op}\ (t_1, t_2)$:
$$\frac{t_1, t_2 \text{ defined} \quad \wedge \quad t \text{ undefined}}{\langle \rho,\ \sigma\rangle \ \to\ \langle \rho[t \mapsto op(\rho[t_1], \rho[t_2])],\ \sigma\rangle}$$

$t_{true}, t_{false} = \textsf{switch}\ (b, t)$:
$$\frac{\rho[b] = \mathit{true} \quad \wedge \quad t \text{ defined} \quad \wedge \quad t_{true}, t_{false} \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[t_{true} \mapsto \rho[t]],\ \sigma\rangle}$$

$t_{true}, t_{false} = \textsf{switch}\ (b, t)$:
$$\frac{\rho[b] = \mathit{false} \quad \wedge \quad t \text{ defined} \quad \wedge \quad t_{true}, t_{false} \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[t_{false} \mapsto \rho[t]],\ \sigma\rangle}$$

$t = \textsf{merge}\ (t_1, t_2)$:
$$\frac{t_1 \text{ defined} \quad \wedge \quad t, t_2 \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[t \mapsto \rho[t_1]],\ \sigma\rangle}$$

$t = \textsf{merge}\ (t_1, t_2)$:
$$\frac{t_2 \text{ defined} \quad \wedge \quad t, t_1 \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[t \mapsto \rho[t_2]],\ \sigma\rangle}$$

$v, d_{out} = \textsf{load}\ (x, d_{in})$:
$$\frac{d_{in}, \sigma[x] \text{ defined} \quad \wedge \quad v, d_{out} \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[v \mapsto \sigma[x],\ d_{out} \mapsto \$],\ \sigma\rangle}$$

$d_{out} = \textsf{store}\ (x, v, d_{in})$:
$$\frac{v, d_{in} \text{ defined} \quad \wedge \quad d_{out} \text{ undefined}}{\langle \rho,\ \sigma\rangle \to \langle \rho[d_{out} \mapsto \$],\ \sigma[x \mapsto \rho[v]]\rangle}$$

Transition Rules for Loop Operators

$t = \textsf{loop}\ (t_{in}, t_{back})$:
$$\frac{t_{in}.I \text{ defined} \quad \wedge \quad t.I.1 \text{ undefined}}{\langle \rho,\ \sigma\rangle \ \to\ \langle \rho[t.I.1 \mapsto \rho[t_{in}.I]],\ \sigma\rangle}$$

$t = \textsf{loop}\ (t_{in}, t_{back})$:
$$\frac{t_{back}.I.j \text{ defined} \quad \wedge \quad t.I.j{+}1 \text{ undefined}}{\langle \rho,\ \sigma\rangle \ \to\ \langle \rho[t.I.j{+}1 \mapsto \rho[t_{back}.I.j]],\ \sigma\rangle}$$

$t, t_{back} = \textsf{until}\ (b, t_{in})$:
$$\frac{\rho[b.I.j] = \mathit{false} \quad \wedge \quad t_{in}.I.j \text{ defined} \quad \wedge \quad t_{back}.I.j \text{ undefined}}{\langle \rho,\ \sigma\rangle \ \to\ \langle \rho[t_{back}.I.j \mapsto \rho[t_{in}.I.j]],\ \sigma\rangle}$$

$t, t_{back} = \textsf{until}\ (b, t_{in})$:
$$\frac{\rho[b.I.j] = \mathit{true} \quad \wedge \quad t_{in}.I.j \text{ defined} \quad \wedge \quad t.I \text{ undefined}}{\langle \rho,\ \sigma\rangle \ \to\ \langle \rho[t.I \mapsto \rho[t_{in}.I.j]],\ \sigma\rangle}$$

Figure 6: Transition Rules for dependence flow graphs

loop. It is sometimes convenient to write this as $t.I$ where I is a (two dimensional) *index vector* $i.j$. To avoid introducing more notation, we will let the term dependence stand for both an identifier (arc) in the dependence flow graph and its dynamic instance in various iterations, relying on context to make the distinction clear. For any index vector I, the add operator can execute as soon as its operands are available, i.e. as soon as $t_1.I$ and $t_2.I$ are defined. Therefore, the rule for the add becomes:

Definition	Precondition	Result
$t = \mathsf{add}\ (t_1, t_2)$	$t_1.I, t_2.I$ defined $\wedge$ $t.I$ undefined	$\langle\rho,\ \sigma\rangle \rightarrow \langle\rho[t.I \mapsto (\rho[t_1.I] + \rho[t_2.I])],\ \sigma\rangle$

To reflect this intuition, the definition of environments is modified:

Definition 2 *Let Seq be the set of finite sequences of positive integers, including the empty sequence. An* environment $\rho : DS \rightarrow V$ *is a finite function — its domain* $DS \subset Dep \times Seq$ *is finite.*

The transition rules for the other operators shown in Figure 6 are extended in a similar manner.

We now discuss the semantics of the loop and exit operators. Figure 7 shows a simple loop and its dependence flow graph. In the first iteration, the statement x := x+1 reads the value of x assigned by the statement x := 1 outside the loop. Therefore, in the dependence flow graph, we must have a dependence from the assignment statement outside the loop to the use within the loop. In subsequent iterations, the statement x := x+1 reads the value of x assigned in the previous iteration. Therefore, in the dependence flow graph, we must have a dependence from the assignment to x in the i^{th} iteration to the use of x in the $(i+1)^{th}$ iteration. The loop operator (see Figure 6) accomplishes this transfer of dependence from outside the loop into the first iteration and from one iteration to the next. The until operator determines if another iteration of the loop should be executed. In the definition $t, t_{back} = \mathsf{until}\ (b, t_{in})$, if b is false then another iteration is to be performed, and the dependence t_{back} is satisfied. Otherwise, the loop is to be exited; if this is the $I.j$ iteration of the loop, the dependence $t.I$ outside the loop is satisfied.

3.3 Properties of the Transition System

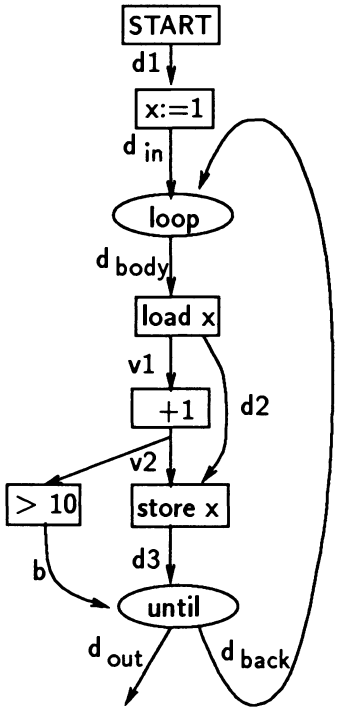
Next, we prove that the transition system for dependence flow graphs, as described above is deterministic in the sense that it has the *one-step Church-Rosser property.* Our proof rests on the fact that dependence flow graphs, by construction, satisfy Bernstein's conditions [Ber66]. This section can be skipped without loss of continuity.

Theorem 1 *If C_0 is a configuration and there exist configurations C_1 and C_2 such that $C_0 \rightarrow C_1$ and $C_0 \rightarrow C_2$, then there is a configuration C_3 such that $C_1 \rightarrow C_3$ and $C_2 \rightarrow C_3$.*

To prove the one-step Church-Rosser property, we first prove that environments grow monotonically during program execution.

```
x := 1;
loop
      x := x+1
until (x > 10)
```

(a) Source Program



(b) Dependence Flow Graph

Figure 7: Dependence flow graph of a simple loop

Definition 3 *If ρ_1 and ρ_2 are environments, then we say that $\rho_1 \subseteq \rho_2$ if the domain of ρ_1 is a subset of the domain of ρ_2 and for all dependencies $t.I$ in the domain of ρ_1, $\rho_1[t.I] = \rho_2[t.I]$.*

Lemma 1 *If $\langle \rho_1, \sigma_1 \rangle \rightarrow \langle \rho_2, \sigma_2 \rangle$, $\rho_1 \subseteq \rho_2$.*

This is evident from inspecting the transition rules in Figure 6. Next, we show that performing a transition cannot disable any other transition.

Lemma 2 *Let τ_1: $C_0 \rightarrow C_1$ be a transition out of configuration C_0, let τ_2 be any other transition out of C_0, and let p_2 be the precondition of τ_2. If $t.I$ is undefined in C_0 but defined in C_1, then (i) p_2 must not imply the condition "$t.I$ undefined", and (ii) p_2 must be satisfied in configuration C_1.*

Proof: (i) Suppose p_2 implies "$t.I$ undefined." Then, since $t.I$ is defined by τ_1, and a dependence has exactly one source, $t.I$ cannot become defined in transition τ_2. Therefore, τ_2 leaves $t.I$ undefined. Examining the rules, we see that only the **merge** operator requires $t.I$ to be undefined and leaves it undefined. However, since p_2 is true in C_0, then the other input to the **merge** must be satisfied in C_0. By Lemma 1, this means that both inputs to the **merge** are satisfied in C_1, which is impossible by construction.

(ii) Except for the load operator, the preconditions in the transition rules of all operators involve only dependencies. If precondition p_2 implies "$t.I$ undefined", then $t.I$ is defined in C_0 and hence in C_1 by Lemma 1. If p_2 implies "$t.I$ undefined", then we have just shown that $t.I$ cannot be defined in C_1. Therefore, if p_2 is an expression involving only dependencies, p_2 is satisfied in C_1.

The precondition of the load operator is the only one that uses the store — it requires that $\sigma[x]$ is defined. If $\sigma[x]$ is defined in C_0, it must be defined in C_1.

Therefore, we conclude that p_2 must be true in C_1. □

Theorem 2 *If C_0 is a configuration and there exist configurations C_1 and C_2 such that $C_0 \to C_1$ and $C_0 \to C_2$, then there is a configuration C_3 such that $C_1 \to C_3$ and $C_2 \to C_3$.*

Proof: Lemma 2(ii) says that the transitions can be performed in either order. We must show that the final store and environment are the same, regardless of the order in which these transitions were performed. From Bernstein's conditions, if one of the transitions performs a store into location x, the other transition cannot read from or store into x. Therefore, the final store is the same regardless of the order of the transitions. From Lemma 1 and 2(i), we conclude that the final environment must be the same as well. □

3.4 Discussion

We can exploit the one-step Church-Rosser property to define a simple interpreter for dependence flow graphs. The interpreter maintains an environment and a store, and keeps a worklist of definitions that may be ready for execution. The initial environment and store are empty, and the worklist is initialized to {START}. While the worklist is not empty, the interpreter dequeues any definition, checks the precondition and performs the transition if the precondition is satisfied. All definitions that are sinks for dependencies sourced by the definition just executed are then enqueued onto the work-list.

The major difference between our representation and conventional dependence graphs is that dependencies, for us, are part of the computational model, and are manipulated by an algebra of operators. Note that load and store, switch and merge, and loop and until are, in an algebraic sense, inverses of each other. Data dependencies are combined with control, and in the next section, we demonstrate how this facilitates the development of optimization algorithms.

4 Constant Propagation on Dependence Flow Graphs

In this section, we demonstrate how global constant propagation can be performed on dependence flow graphs using a simple algorithm based on abstract interpretation. We show that for any program, we can write down a set of equations whose solution corresponds to the possible-paths constants for that program. Next, we show that this solution can be computed efficiently, thereby developing an algorithm that has the same asymptotic complexity as the algorithm due to Wegman and Zadeck [WZ84]. Finally, we give a proof of correctness for our algorithm.

4.1 Equational Characterization of Constants

Figure 8 shows how to write down a set of semantic equations from a dependence flow graph representation of a program. The equations are obtained by replacing the operator in each definition with a function that denotes the abstract interpretation of the operator in *Lat*. We let *DenOp* stand for the interpretation of operation **op** in the domain *Lat*. For example, the function *DenSw* stands for the interpretation of **switch** and is defined as follows:

$$DenSw(b, c) = \begin{cases} c, c & \text{if } b = \top \\ c, \perp & \text{if } b = \textsf{true} \\ \perp, c & \text{if } b = \textsf{false} \\ \perp, \perp & \text{if } b = \perp \end{cases}$$

This is similar to the standard interpretation of **switch**, except that it deals with the case when b is $\top$ — if the value of the predicate cannot be determined during constant propagation, then the input value c is propagated to both sides of the **switch**. Therefore, in the abstract interpretation, both inputs of a **merge** can be defined. The output of a **merge** operator is constant only if both inputs are the same constant or if one side of the **merge** is never executed, and the other side is constant. In the equations, this is stated compactly using the least upper bound operator on *Lat*.

In the standard semantics, the global store was used to communicate values between the **load** and **store** operators. As is the case in all the algorithms discussed in Section 2, the global store plays no role in our constant propagation algorithm. Instead, we take advantage of the fact that there is a flow dependence path in the graph from a **store** to every **load** dependent on it. Lattice values are propagated along these paths. The **store** operator propagates the value of its input to its output, provided that d_{in} is not $\perp$ — that is, if it is possible that this operator may be executed. Therefore, the **load** operator need only propagate the value of input d_{in} to its outputs. It is possible to define a more sophisticated interpretation of **until** that checks if the boolean b is constant, thereby detecting non-terminating loops as well as loops that execute exactly once.

For any dependence flow graph, we can write down a set of semantic equations over *Lat*. In these equations, each identifier occurs on the left hand side of exactly one equation and the functions on the right-hand side of the equations are monotonic and continuous. It is

SYNTACTIC EQUATION		SEMANTIC EQUATION	
d	$=$ START ()	d	$= \$$
v	$=$ op (v_1, v_2)	v	$= DenOp(v_1, v_2)$
v_t, v_f	$=$ switch (b, v)	v_t, v_f	$= DenSw(b, v)$
v	$=$ merge (v_1, v_2)	v	$= v_1 \sqcup v_2$
v, d	$=$ load (x, d_{in})	v, d	$= d_{in}, d_{in}$
d	$=$ store (x, v, d_{in})	d	$=$ if $d_{in} = \perp$ then $\perp$ else v
v	$=$ loop (v_{in}, v_{back})	v	$= v_{in} \sqcup v_{back}$
v, v_{back}	$=$ until (b, v_{in})	v, v_{back}	$= v_{in}, v_{in}$

Figure 8: Abstract Interpretation of Operators

a well-known result that such a system of equations has a least solution [Man81]. Figure 9 shows these values for the program of Figure 5. The possible-paths constants can be read off from the least solution of the semantic equations as follows. Let $\mathcal{C} : D \to Lat$ be the least solution. If t is a functional dependence, then, as in Section 2, $\mathcal{C}[t] = \perp$ means that the operator that defines t will never be executed, $\mathcal{C}[t] = c$ means that if t is ever assigned a value, it is assigned the value c , and $\mathcal{C}[t] = \top$ means that the value of t cannot be determined to be constant. The values assigned to imperative dependencies must be interpreted a little differently. Consider dependence d4 in Figure 9. This dependence is given the value 1 by the constant propagation algorithm, but it is given the value $ in the standard interpretation which uses the global store, rather than dependencies, to transmit values between a **store** operation and corresponding load operations. If t is an imperative dependence, then $\mathcal{C}[t] = \perp$ means that the operator that defines t will never be executed, but if $\mathcal{C}[\mathrm{t}]$ is any other value, we conclude that this operator may be executed — that is, t may get the value $ in the standard interpretation.

To compute the least solution of the equations efficiently, we run the dependence flow graph interpreter defined in Section 3.4, using the abstract, rather than the standard, interpretation of the operators. This abstract interpreter maintains only an environment, since the store plays no role in constant propagation. In this environment, we keep only a single value associated with each dependence, rather than an indexed family of values, because a dependence is constant only if it is constant in all iterations. Since there are only a finite number of dependencies, we start with an initial environment that maps all dependencies to $\perp$. The worklist of definitions ready for processing is initialized to {START}. While the worklist is non-empty, we remove a definition from the worklist and update the environment using the abstract interpretation rules shown in Figure 8. This update to the environment consists of binding new values to the dependencies whose source is the definition being interpreted.

If the value bound to a dependence changes, we add every definition that is a sink for that dependence to the worklist. Since there can be $O(EV)$ edges in the dependence flow graph and the value propagated along each edge can change at most twice, the complexity of this constant propagation algorithm on dependence flow graphs is $O(EV)$. This is the same asymptotic complexity as that of the algorithm due to Wegman and Zadeck.

As an aside, we note that it is possible to express this algorithm as a state transition system like the one defined in Figure 6. For example, if $\hat{\rho}$ is the environment maintained for constant propagation, the transition rule for op is the following:

$$t = \mathsf{op}\ (t_1, t_2) : \qquad \frac{DenOp(\hat{\rho}[t_1], \hat{\rho}[t_2]) \sqsupset \hat{\rho}[t]}{\hat{\rho} \to \hat{\rho}[t \mapsto DenOp(\hat{\rho}[t_1], \hat{\rho}[t_2])]}$$

The transition rules for the other operators can be written down similarly. We have chosen to present the equational characterization since the essence of constant propagation is clearer in that framework.

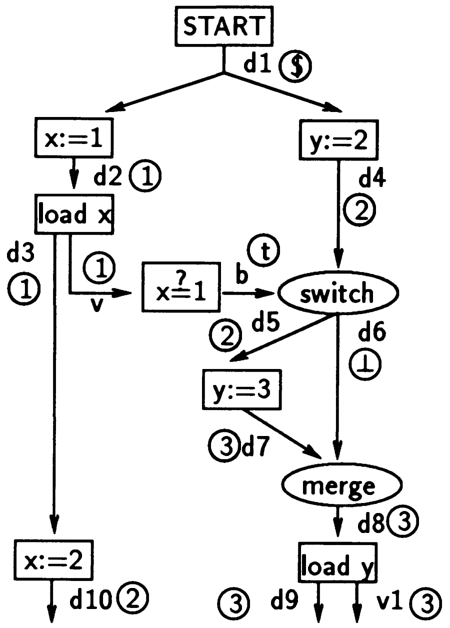


Figure 9: Result of Constant Propagation in Figure 1(d)

4.2 Correctness of Constant Propagation

The intuitive idea behind the proof of correctness is the following. We use an induction on the length of the execution sequence of the program. Imagine that each dependence in the dependence flow graph is labeled with the value from *Lat* that is assigned to it by the constant propagation algorithm. We run the standard interpretation, and each time a (syntactic) value is assigned to a dependence, we make sure that that value is "consistent" with the value (in *Lat*) determined by the constant propagation algorithm. To define what consistent means, we need two definitions.

1. We need a way to compare syntactic values with values in *Lat*. Therefore, corresponding to a syntactic environment ρ, we associate the natural semantic environment $\hat{\rho}$ such that $\hat{\rho}[t.I]$ is $\perp$ if $t.I$ is undefined in ρ and $\hat{\rho}[t.I]$ is a boolean or integer constant in *Lat* if $\rho[t.I]$ is that syntactic constant.

2. The constant propagation algorithm uses imperative dependencies to pass values between store's and load's, but in the standard interpretation, these dependencies carry the value **\$**. Therefore, if $\mathcal{C} : D \rightarrow Lat$ is the result of constant propagation, we define a function $\mathcal{C}^* : D \rightarrow Lat$ such that $\mathcal{C}^*[t]$ is the same as $\mathcal{C}[t]$ unless t is an imperative dependence mapped to a non-$\perp$ value by $\mathcal{C}$ in which case $\mathcal{C}^*[t]$ is **\$**.

If ρ_f is the final environment produced by the standard interpreter, then the constant propagation algorithm is correct if $\forall t\ \forall I.\hat{\rho}_f[t.I] \sqsubseteq \mathcal{C}^*[t]$. In other words,

1. if $\mathcal{C}^*[t] = \bot$, that is, constant propagation determines t is never assigned a value in any execution, then t must not be assigned in this execution; i.e., $\mathcal{C}^*[t] = \bot \Rightarrow \forall I.\hat{\rho}_f[\text{t.I}] = \bot$.

2. if $\mathcal{C}^*[t] = c$ where c is a constant, then every assignment to t in this execution must assign c; i.e., $\mathcal{C}^*[t] = c \Rightarrow \forall I.\hat{\rho}_f[t.I] = \bot$ (t never assigned) or $\hat{\rho}_f[t.I] = c$.

3. if constant propagation cannot determine that t is dead or constant in all executions, then nothing can be said about the value assigned to t; i.e. $\mathcal{C}^*[t] = \top \Rightarrow \forall I.\hat{\rho}_f[t.I] \in Lat$.

Theorem 3 *If ρ is the environment at any point in the execution, then $\forall t\ \forall I.\hat{\rho}[t.I] \sqsubseteq \mathcal{C}^*[t]$.*

Proof:

Basis: Vacuously true, since the initial environment for the standard interpreter is the empty environment.

Induction step: Suppose that the hypothesis holds at some point in the execution, and the interpreter executes some definition.

1. Suppose the definition is $v_{out}, d_{out} = \mathsf{load}(x\ , d_l)$, and that some definition s of the form $d_s = \mathsf{store}(x, v_s, d)$ is the corresponding **store** that stored the value read by this **load**. There must be a path of satisfied flow dependencies from s to the **load**, and any intermediate nodes on this path must be from the set {**switch, merge, loop, until**}. Now, if I_l and I_s are the relevant index vectors for the **load** and **store**,

   - $\mathcal{C}[v_s] = \mathcal{C}[d_s]$ (from Figure 8 and inductive hypothesis)
   - $\mathcal{C}[d_s] \sqsubseteq \mathcal{C}[d_l]$ (since intermediate nodes are {**switch, merge, loop, until**})
   - $\mathcal{C}[d_l] = \mathcal{C}[v_{out}]$ (from Figure 8)
   - Therefore, $\mathcal{C}[v_s] \sqsubseteq \mathcal{C}[v_{out}]$
   - $\hat{\rho}[v_{out}.I_l] = \hat{\rho}[v_s.I_s]$ (**store** corresponding to **load**)
   - $\hat{\rho}[v_s.I_s] \sqsubseteq \mathcal{C}[v_s]$ (from inductive hypothesis)
   - Therefore, $\hat{\rho}[v_{out}.I_l] \sqsubseteq \mathcal{C}[v_{out}] = \mathcal{C}^*[v_{out}]$
   - Also, $\hat{\rho}[d_{out}.I_l] = \$ = \mathcal{C}^*[d_{out}]$

   Since the environment before and after interpretation of the **load** differ only in the binding of dependencies $v_{out}.I_l$ and $d_{out}.I_l$, we conclude that the inductive hypothesis holds after this step.

2. For any other instruction, the output is a function of the inputs and is independent of the global store.

   Let $V_o = F(V_i)$ be the syntactic definition processed in this step. By the induction hypothesis, $\hat{\rho}[V_i.I_i] \sqsubseteq \mathcal{C}^*[V_i]$. By examining each case in Figures 6 and 8, it is easy to verify that if ρ_1 is the environment after the instruction is executed, then $\hat{\rho}_1[V_o.I_o] \sqsubseteq \mathcal{C}^*[V_o]$.

By induction, $\forall t\ \forall I. \hat{\rho}_f[t.I] \sqsubseteq \mathcal{C}^*[t]$.

□

5 Conclusions

An interesting aspect of dependence flow graphs is that they exhibit many features of previously proposed representations. These connections will be discussed in full in another paper — here, we summarize them. Connections with data and program dependence graphs have already been discussed in Section 2.

Continuation passing style (CPS) is an executable representation that was proposed by Steele and Sussman as a suitable intermediate language for compiling Scheme [SS78]. Since then, it has been used in a number of other compilers such as the Standard ML compiler [AM87]. Informally, the continuation of an operator is a representation of the effect of the rest of the program after the operator is executed. The execution model of CPS is sequential. Arcs in dependence flow graphs can be viewed as continuations in a *parallel* execution model. This generalization is crucial since it lets us represent dependence information directly in the representation. However, in CPS, continuations are first-class citizens in the sense that they can be passed into and out of functions. As of now, we do not permit that in our model, but we intend to do so when we extend our work to the inter-procedural level.

Static single assignment (SSA) is a compact representation of def-use chains that uses so-called ϕ-functions to combine def-use arcs [CFR$^+$89]. ϕ-functions are similar to our **merge** nodes and we get the same compactness advantage in our representation. In addition, unaliased variables are renamed so that each variable is assigned by just one statement. The result of this renaming is to assign a unique name to each dependency, which is convenient for program transformation. In our representation, the unique naming of dependencies is fundamental, and not a variant used for optimizations. If necessary, updatable storage locations can be eliminated in a dependence flow graph through a simple program transformation — in essence, imperative dependencies are converted into functional dependencies [BP90].

In translating the dataflow language VAL into static dataflow graphs, Ackerman defined an intermediate representation called VAL program graphs [Ack84]. This representation has become popular in the dataflow world; with minor modifications, it has been used by Traub to translate the dataflow language Id into dynamic dataflow graphs [Tra86]. However, this representation supports only functional languages, and cannot handle imperative updates or arbitrary flow of control as dependence flow graphs can.

At HP Labs, Rau and Schlansker are investigating an intermediate form called PIF (parallel intermediate form) for compiling FORTRAN to VLIW machines.[9] Ekanadham at IBM has an intermediate form called Kudos which is used to translate FORTRAN and functional languages into code for the Empire hybrid dataflow machine [Eka90]. Suhler at IBM, and Ballance, Maccabe and Ottenstein at Los Alamos, have been working on implementing FORTRAN on dataflow machines. In an earlier paper [BP89], we solved this problem completely, and pointed out the advantages of basing intermediate languages on the dependence-driven execution model. While our suggestion has been taken to heart by these researchers [BMO90], it is too early to tell if there will be a convergence of these ongoing efforts.

[9] Personal communication.

Future Research

The ideas presented in this paper form the basis of the Typhoon parallelizing compiler project at Cornell University. We have implemented prototype compilers that translate programs in FORTRAN and in the dataflow language Id into an intermediate language called Pidgin. Pidgin is a textual form of the dependence flow graph data structure, on which our optimizer is built. The optimizing portion of the compiler is a source-to-source transformer of Pidgin programs.

Currently, our optimizer implements the constant propagation algorithm presented in this paper, as well as other standard optimizations including constant subexpression elimination, dead code elimination, and loop invariant code motion. We have implemented an interpreter for Pidgin that can generate a profile of dynamic program parallelism. Work in progress includes the development of new algorithms for global redundancy elimination and strength reduction based on abstract interpretation.

Much of our future work will focus on architecture-driven loop transformations and scheduling and code generation for specific architectures. We have extended the definition of dependence flow graphs so that we can represent the dependence information needed to implement loop transformations. Preliminary work on the scheduling of dependence flow graphs has focused on generating code for pipelined RISC architectures such as SPARC; future work will target dataflow architectures, Non-Uniform Memory Access machines and Very Long Instruction Word architectures.

Acknowledgements

We are indebted to Radha Jagadeesan for catching several errors in the proofs. It is a pleasure to acknowledge the support and encouragement of Bob Rau and Mike Schlansker of the Concurrent Computing Department at HP. We have benefited from discussions with them and with Corky Cartwright at Rice and Ekanadham at IBM. The persistent questioning of Arvind and Nikhil at M.I.T. has done much to sharpen our presentation. Charles DeVane, Richard Huff, Wei Li and Anne Rogers have provided useful comments.

References

[AA89] Zena Ariola and Arvind. PTAC: A parallel intermediate language. In *Proceedings of the Functional Programming Languages and Computer Architecture*, London, September 1989.

[Ack84] W. B. Ackerman. Efficient implementation of applicative languages. Technical Report TR-323, M.I.T. Laboratory for Computer Science, April 1984.

[AH82] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 17(6):22–31, June 1982.

[AM87] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Lecture Notes In Computer Science*, September 1987.

[ANP89] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11, 1989.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Computers*, 1(5):757–762, October 1966.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *Proceedings of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, 25(6):257–271, June 1990.

[BP89] M. Beck and K. Pingali. From control flow to dataflow. Technical Report TR89-1050, Cornell University, October 1989.

[BP90] Micah Beck and Keshav Pingali. From control flow to dataflow. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[CC77] P. Cousout and R. Cousout. Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, January 1977.

[CC79] P. Cousout and R. Cousout. Systematic design of program analysis frameworks. *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[CF89] R. Cartwright and M. Felleisen. The semantics of program dependence. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 25(6), June 1989.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation, Vol. 19, Lecture Notes in Computer Science*, pages 362–376, 1974.

[Eka90] K. Ekanadham. Kudos. IBM Yorktown Heights, 1990.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependency graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, 1978.

[Man81] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Publishing Company, 1981.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, pages 1184–1201, December 1986.

[RL77] John H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 104–118, January 1977.

[SS78] G. Steele and G. Sussman. RABBIT: a compiler for SCHEME. Technical Report AI memo 474, M.I.T. Laboratory for Artificial Intelligence, May 1978.

[Tra86] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical report, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, August 1986.

[WZ84] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 291–299, 1984.