Dependence Analysis for Pointer Variables

Susan Horwitz, Phil Pfeiffer, and Thomas Reps
University of Wisconsin–Madison

Abstract

Our concern is how to determine data dependences between program constructs in programming languages with pointer variables. We are particularly interested in computing data dependences for languages that manipulate heap-allocated storage, such as Lisp and Pascal. We have defined a family of algorithms that compute safe approximations to the flow, output, and anti-dependences of a program written in such a language. Our algorithms account for destructive updates to fields of a structure and thus are *not* limited to the cases where all structures are trees or acyclic graphs; they *are* applicable to programs that build cyclic structures.

Our technique extends an analysis method described by Jones and Muchnick that determines an approximation to the actual layouts of memory that can arise at each program point during execution. We extend the domain used in their abstract interpretation so that the (abstract) memory locations are labeled by the program points that set their contents. Data dependences are then determined from these memory layouts according to the component labels found along the access paths that must be traversed during execution to evaluate the program's statements and predicates.

For structured programming constructs, the technique can be extended to distinguish between loop-carried and loop-independent dependences, as well as to determine lower bounds on minimum distances for loop-carried dependences.

1. INTRODUCTION

A number of different kinds of data dependences have been identified, including *flow* dependences, *anti*-dependences, and *output* dependences. Program point q has a dependence on program point p due to location *loc* if execution can reach q after p and there is no intervening write into *loc* along the execution path by which q is reached from p. There is a flow dependence if p writes into *loc* and q reads *loc*, an output dependence if p and q both write into *loc*, and an anti-dependence if p reads *loc* and q writes into *loc*.

Determining data dependences between program statements is crucial to automatic program parallelization and vectorization [2, 6, 14] as well as to automatic program integration [8]. Past work has provided techniques for determining data dependences for languages with scalar variables and arrays; our concern in this paper is how to determine data dependences for languages that have pointer-valued variables and heap-allocated storage (*e.g.* Lisp and Pascal). We describe a family of methods that compute safe approximations to a program's data dependences. The methods account for destructive updates to fields of a structure and are *not* limited to simple cases where all structures are trees or acyclic graphs; they *are* applicable to programs that build up cyclic structures.

Although our presentation concentrates on the computation of flow dependences, the methods we give can be extended to compute anti-dependences and output dependences. For structured programming constructs, the method can also be extended to distinguish between loop-carried and loop-independent dependences, as well as to determine lower bounds on minimum distances for loop-carried dependences. (These extensions are described in Sections 4 and 5.)

For scalar variables, one way to compute flow dependences involves first computing a more general piece of information: the set of *reaching definitions* for each program point [1, 13]. A definition of variable x at program point p *reaches* point q if there is an execution path from p to q such that no other definition of x appears on the path. The set of reaching definitions for a program point q is the set of program points that generate definitions that reach q. Program point q is *flow dependent* on all members of the reaching-definition set that define variables used at q.

To extend the concept of flow dependence for languages that manipulate heap-allocated storage, it is necessary to phrase the definition in terms of *memory locations* rather than *variables*:

> Program point q has a flow dependence on program point p if p writes into a memory location *loc* that q reads, and there is no intervening write into *loc* along the execution path by which q is reached from p.

Unlike the situation that exists for programs with (only)

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and DCR-8603356, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, and Xerox.

Authors' address: Computer Sciences Dept., University of Wisconsin – Madison, 1210 W. Dayton St., Madison, WI 53706.

scalar variables, where there is a fixed "layout" of memory, for programs that manipulate heap-allocated storage not all accessible memory locations are named by program variables. In the latter situation new memory locations are allocated dynamically in the form of cells taken from the heap.

To compute data dependences between constructs that manipulate and access heap-allocated storage, our starting point is the method of Jones and Muchnick, described in [12], which, for each program point q, determines a set of structures that approximate the different "layouts" of memory that can possibly arise at q during execution. We extend the domain employed in the Jones-Muchnick abstract interpretation so that the (abstract) memory locations are labeled by the program points that set their contents. Flow dependences are then determined from these memory layouts according to the component labels found along the access paths that must be traversed during execution to evaluate the program's statements and predicates.

Our method for computing flow dependences is presented as an abstract interpretation of programs in an imperative language without procedure calls[1] that has the following features: (1) variables can point to heap-allocated storage, (2) storage is allocated by expressions of the form $cons(exp, exp)$, (3) storage is accessed by expressions such as x.hd and x.tl, and (4) destructive updating is accomplished by statements such as $x.\mathrm{hd} := 0$, which updates the hd field of the cell pointed to by x. We further assume that execution flow of control can be captured in a standard control-flow graph.

The remainder of the paper is organized in five sections. Section 2 presents an informal summary of our results. Section 3 develops an algorithm for computing a safe approximation to a program's flow dependences. Section 4 describes algorithms for computing a program's output and anti-dependences. Section 5 describes techniques for analyzing how dependences are carried by a program's loops. Section 6 concludes with a discussion of related work.

2. SUMMARY OF RESULTS

This paper examines the concept of flow dependence using four related semantic definitions, which we refer to as the standard semantics, the instrumented semantics, the collecting semantics, and the abstract semantics. For each semantics, we give a different characterization of flow dependence. We show that the characterizations for the standard semantics, the instrumented semantics, and the collecting semantics are equivalent, and that the characterization of flow dependence for the abstract semantics is a safe approximation to the other three. The latter characterization provides an algorithm for computing an approximation to (*i.e.* a superset of) a program's flow dependences.

Our use of *four* semantics is somewhat at variance with the framework for abstract interpretation laid out in [4], which involves the use of *three* semantics: a standard semantics, a collecting semantics, and an abstract semantics. In our treatment, the instrumented, collecting, and abstract semantics do follow the model of [4]; however, the instrumented semantics is not what one would call a *standard* semantics. In developing a dependence-analysis technique using the abstract-interpretation paradigm, one of our concerns about starting with the instrumented semantics was that the extra information that is part of the instrumented semantics could restrict the applicability of the analysis algorithm to just a particular class of language implementations; that is, *the instrumented semantics could embody an implicit implementation commitment*. We were able to put these fears to rest by proving that, for the purpose of determining flow dependences, the standard semantics and the instrumented semantics are equivalent.

Because data dependences are relations between program points, characterizing dependences requires reasoning about a program's intensional (internal) behavior as well as its extensional (external) behavior. Therefore, the standard semantics is given as an operational semantics, which describes a program's internal state; execution is defined as a sequence of transformations over (*program-point, store*) pairs. For the standard semantics, a flow dependence is defined in terms of a program's behavior over a sequence of states; there is a flow dependence from point p to point q, denoted by $p \longrightarrow_f q$, if it is possible to have a sequence of states in which q reads a location last written by p.

The instrumented semantics is an alternative semantics that permits dependences to be characterized in terms of an *individual* state instead of a *sequence* of states. The term "instrumented" suggests how the instrumented semantics relates to the standard semantics: in the instrumented semantics each location in the (instrumented) store is annotated with the program point that last wrote to the location. We redefine $p \longrightarrow_f q$ to mean that there exists a state such that program point q reads a location annotated with program point p; we then prove the latter definition equivalent to the original definition.

Having defined flow dependence in terms of states that can be generated by the transition function of the instrumented semantics, we define a *collecting semantics* in which a program's meaning is the set of all non-error states that would be generated by the transition function. We then redefine $p \longrightarrow_f q$ in terms of the collecting semantics and prove the new definition equivalent to the earlier definitions.

[1]The method can be extended to handle procedure calls; a straightforward, albeit conservative, approach involves treating all paths in the call graph as possible execution paths.

The preceding characterizations do not provide an effective procedure for computing flow dependences; therefore, we define an abstract semantics that associates with each program point a set of graphs, called *store graphs*, that approximate the set of (instrumented) stores that could exist at that point during program execution. Each variable used in the program has a corresponding vertex in the graph. Other vertices represent memory locations allocated (in pairs) during execution; pairs of vertices correspond to *cons* cells. Each vertex is labeled with the program point that most recently wrote into the location represented by the vertex.

The set of store graphs associated with each program point approximates the set of stores that could exist at that point during program execution. The approximation is necessary whenever the program includes a loop. As discussed in Section 3.4, three kinds of approximations are used so that an individual store graph can approximate a potentially infinite set of stores.

Flow dependences are determined by examining a program point's expressions in conjunction with its set of store graphs. A program point is potentially flow dependent on all program points that most recently wrote into a location read during expression evaluation. We show that the data dependences determined in this fashion are a superset of the actual data dependences that exist according to the preceding three characterizations.

3. FROM STANDARD TO ABSTRACT SEMANTICS

Section 3.5 gives an algorithm that computes a safe approximation to a program's flow dependences. Sections 3.1 through 3.4 derive this algorithm by constructing a progression of four semantics. The standard semantics, presented in Section 3.1, yields a formal characterization of flow dependence that resembles the definition given in the introduction. The instrumented semantics, presented in Section 3.2, yields an equivalent characterization of flow dependence based on how programs affect *individual* states. The collecting semantics, presented in Section 3.3, yields an equivalent characterization of flow dependence based on an alternative definition of program behavior. The collecting semantics is used to develop the abstract semantics's characterization of flow dependence given in Section 3.4 — the characterization that forms the basis of our algorithm.

3.1. The Standard Semantics

In the standard semantics, a *program* is a finite collection of program points, characterized by a *successor function* that maps program points to their control-flow successors. *Point*, the domain of program points, contains two distinguished values, **initial** and **final**.

A *computation* is a sequence of states generated by the transition function *next*:

$$
\begin{aligned}
&next: \ Prog \to State_\perp \to State_\perp \\
&Prog = Point \times Bool \to Point \\
&State = Point \times Store \\
&Store = Loc \to Val + \{\,\text{UNUSED}\,\} \\
&Val = \ Atom + (Loc \times Loc)
\end{aligned}
$$

Atom is a space of atoms, *e.g.* integers, that contains the distinguished value **?** (*i.e., "uninitialized"*).

An *environment* is a finite map from identifiers to locations. We have simplified our equations by treating the environment as a global variable; that is, the *next* function uses a predetermined, read-only environment to interpret the meaning of identifier expressions.

Using *fix* to denote the least-fixpoint functional, the program-meaning function **M** is defined as

$$
\begin{aligned}
&\mathbf{M}: Prog \to Store \to Store_\perp \\
&\mathbf{M}(prog) = \lambda store. \\
&\quad \textbf{let } programFunction = \\
&\qquad (fix\ \lambda f.\ (\lambda (point', store'). \\
&\qquad\quad \textbf{if } point' = \textbf{final then } (point', store') \\
&\qquad\quad \textbf{else } f\,(next\,(prog, (point', store'))) \\
&\qquad\quad \textbf{fi})) \\
&\quad \textbf{in} \\
&\qquad \textbf{let } (point'', store'') = programFunction\,(\textbf{initial}, store) \\
&\qquad \textbf{in } store'' \\
&\qquad \textbf{end} \\
&\quad \textbf{end}
\end{aligned}
$$

Our informal definition of a flow dependence from program point p to program point q was that p writes to a location that q reads (and there is no intervening write). We now restate this definition in terms of the standard semantics:

DEFINITION: The *state transition relation* $\cdots \vdash \cdots \longrightarrow \cdots$ is defined as:

$$
\begin{aligned}
&prog \vdash state_i \longrightarrow^n state_j \\
&\qquad \Leftrightarrow n \geq 0 \wedge state_j = (next\,(prog))^n\ (state_i) \\
&prog \vdash state_i \longrightarrow^* state_j \\
&\qquad \Leftrightarrow \exists\, n : prog \vdash state_i \longrightarrow^n state_j \\
&prog \vdash state_i \longrightarrow^+ state_j \\
&\qquad \Leftrightarrow \exists\, n > 0 : prog \vdash state_i \longrightarrow^n state_j \\
&prog \vdash state_n \longrightarrow \cdots \longrightarrow state_m \\
&\qquad \Leftrightarrow \forall\, i : n \leq i \leq m-1 : prog \vdash state_i \longrightarrow state_{i+1}
\end{aligned}
$$

□

DEFINITION: A location is *read* when it is used as a parameter to a *store*. A location is *written* when it is first allocated (as the result of evaluating a *cons* expression) or when it is updated in a store. (A transition out of the initial state reads and writes all locations.) □

DEFINITION: The set *readset* (*prog*, *state*) denotes those locations read during the evaluation of the expression *next* (*prog*, *state*). The set *writeset* (*prog*, *state*) denotes those locations written during the evaluation of the expression *next* (*prog*, *state*). □

DEFINITION: A predicate $P: State \to Bool$ is *true for all states between* $state_n$ *and* $state_m$ iff whenever

$prog \vdash state_n \longrightarrow \cdots \longrightarrow state_m$, $P(state_i) = true$ for all $i : n < i < m$. □

DEFINITION: Let *prog* be a program with program points p and q. Point p *transmits* a value to point q relative to *store* iff there exist stores $store_p$ and $store_q$ such that all of the following hold:

$prog \vdash (\textbf{initial}, store) \longrightarrow^* (p, store_p) \longrightarrow^+ (q, store_q)$;
$loc \in writeset\ (prog, (p, store_p))$;
for all $(point, store)$ between $(p, store_p)$ and $(q, store_q)$,
$loc \notin writeset\ (prog, (point, store))$;
$loc \in readset\ (prog, (q, store_q))$. □

DEFINITION (*Flow Dependence*): For any program *prog* with program points p and q, there is a *flow dependence* from p to q iff there exists a *store* such that p transmits a value to q relative to *store*. □

Example. Figure 1 depicts an example program's computation on an initially empty store. The five graphs labeled $store_0$ through $store_4$ depict memory "layouts" at successive stages in the computation. Rectangles represent memory locations. Pairs of adjoined rectangles represent *cons cells*, *i.e.*, pairs of memory locations allocated by cons operations.

Rectangles in Figure 1 are annotated with labels that represent names, identifiers, and values. A location's *name* is a string of the form l_n that distinguishes it from all other locations in a store. A label of the form "*ident*:" indicates that a location (such as l_0) is associated with the specified identifier. A location's *value* is depicted as an atom (*i.e.*, an integer or ?) if the location contains an atomic value, or as an arrow if the location contains a reference to a cons cell.

The concept of a reference to a cons cell is actually an abstraction that makes diagrams of stores easier to read. Every edge of the form $l_1 \longrightarrow (l_2, l_3)$ should be regarded as the *pair* of *labeled* edges $l_1 \xrightarrow{\text{hd}} l_2$ and $l_1 \xrightarrow{\text{tl}} l_3$.

The table shown below the example program's computation in Figure 1 tells which memory locations are read and written at each step in the program's computation. The program's flow dependences are shown in the example program itself and in the table. There is a dependence [1] $\longrightarrow_f$ [2] because point [1] sets the value of x (location l_0), which is read by point [2]. Similarly, there is a dependence [2] $\longrightarrow_f$ [3] because point [2] sets the value of y (location l_1), which is read by point [3] to access y.hd. There is a dependence [1] $\longrightarrow_f$ [3] because point [1] writes x.hd (location l_2), which is the location of y.hd at point [3]. ■

3.2. An Instrumented Semantics

In the previous section, $p \longrightarrow_f q$ was defined in terms of a program's behavior over a sequence of states $(p, store_p) \rightarrow \cdots \rightarrow (q, store_q)$. We now present a second semantics in which flow dependences may be determined by examining states in isolation. We alter the previous semantics by labeling each location with the program point that last defined its value. We then redefine $p \longrightarrow_f q$ to mean that q reads a location labeled p, and prove the new definition equivalent to the definition of Section 3.1. (The subscript "I" is used to distinguish elements of the instrumented semantics from similarly named elements of the standard semantics.)

In the instrumented semantics a computation's state is redefined to be a pair $State_I = Point \times LabeledStore$, where $LabeledStore = Label \times Store$ and $Label = Loc \rightarrow Point$. Members of *Label* pair a location with the point that last defined its value.

The meaning of a computation in the instrumented semantics is defined by $next_I$, a state-transition function for $State_I$. State-transition function $next_I$ is the same as *next* (the state-transition function for the standard semantics), except that $next_I$ must sometimes update labels, as follows: (1) When a transition is made out of the initial state, all labels are set to the distinguished point **initial**. (2) When a transition is made out of a state whose program-point component p corresponds to an assignment statement, the following labels are set to p: (a) the labels of all locations newly allocated as a result of evaluating the right-hand side of the assignment (*i.e.* because of occurrences of *cons* sub-expressions); and (b) the label of the location corresponding to the left-hand side of the assignment.

The program meaning function $\mathbf{M_I}$ is defined as

$\mathbf{M_I}$: $Prog \rightarrow Store \rightarrow LabeledStore_\perp$
$\mathbf{M_I}(prog) = \lambda store.$
let *programFunction* =
($fix\ \lambda f.$
($\lambda (point', (label', store')).$
if $point' =$ **final then** $(point', (label', store'))$
else $f\ (next_I (prog, (point', (label', store'))))$
fi))
in
let $(point'', (label'', store'')) =$
programFunction (**initial**, ($\lambda loc.$**initial**, *store*))
in $(label'', store'')$
end
end

In [10] we prove two lemmas about the relationship between the standard and the instrumented semantics. The first lemma says that (1) every state transition in the standard semantics has a corresponding state transition in the instrumented semantics; (2) the two semantics induce the same readsets; and (3) the instrumented semantics updates label values for exactly those locations in the standard semantics's writeset. The second lemma generalizes the first to sequences of state transitions and demonstrates that a location's label names the last program point that defined its value.

We now define flow dependence relative to the instrumented semantics and state a theorem that says that this definition is equivalent to the corresponding definition in

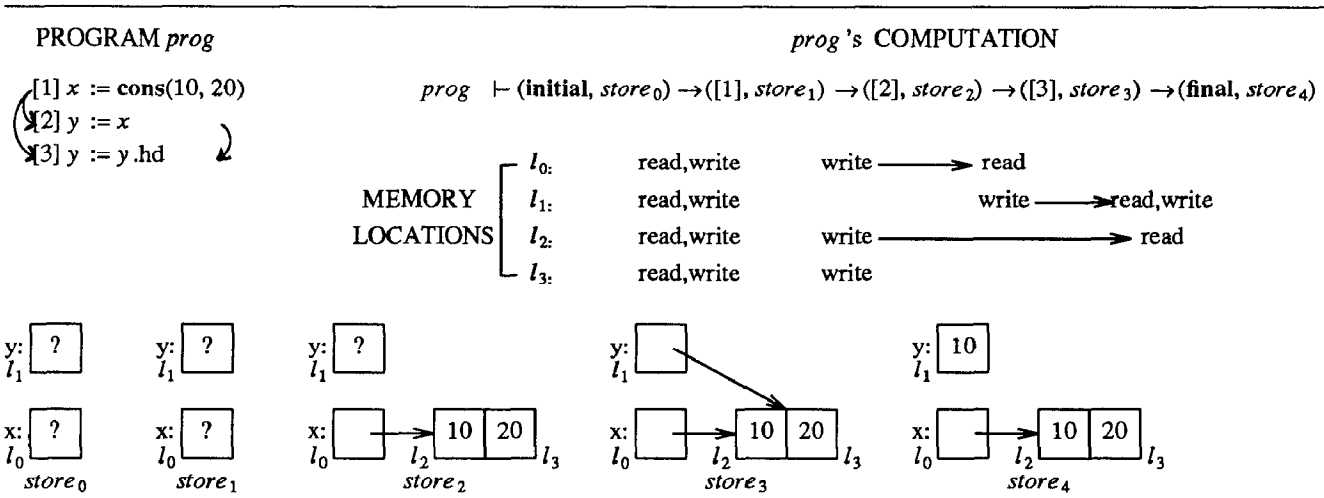

Figure 1. Illustration of the standard semantics and of flow dependence under the standard semantics. The program's flow dependences are shown in the example program itself and in the table. There is a dependence [1] $\longrightarrow_f$ [2] because point [1] sets the value of x (location l_0), which is read by point [2]. Similarly, there is a dependence [2] $\longrightarrow_f$ [3] because point [2] sets the value of y (location l_1), which is read by point [3] to access y.hd. There is a dependence [1] $\longrightarrow_f$ [3] because point [1] writes x.hd (location l_2), which is the location of y.hd at point [3].

the standard semantics.

DEFINITION: Let $(label, store)$ be a member of *LabeledStore*. Point q *reads a location labeled* p in *store* (relative to *prog* and *label*) iff there exists *loc* such that $loc \in readset\ (prog, (q, store))$ and $label\ (loc) = p$. □

DEFINITION (*Flow Dependence relative to the Instrumented Semantics*): For any program *prog* with program points p and q there is a *flow dependence* from p to q iff there exist *store* and $(label_q, store_q) \in LabeledStore$ such that

1. $prog \vdash_{\mathbf{I}} (\mathbf{initial}, (\lambda\, loc.\mathbf{initial}, store)) \longrightarrow^+ (q, (label_q, store_q))$, and
2. q reads a location labeled p in $store_q$ (relative to *prog* and $label_q$). □

THEOREM 1: For any program *prog* with points p and q, $p \longrightarrow_f q$ according to the definition of flow dependence relative to the instrumented semantics iff $p \longrightarrow_f q$ according to the definition of flow dependence relative to the standard semantics.

Example. Figure 2 illustrates the computation of the example program of Figure 1 relative to the instrumented semantics. Flow dependences are determined by examining individual states rather than sequences of states. For example, the flow dependences incident on point [3] are determined by finding all states of the form ([3], $store_i$) and examining the labels in the locations in $store_i$ read by point [3] (namely, the locations that correspond to y and to y.hd). In this example, there is just one such state, the store is $store_3$, the locations are l_1 and l_2, respectively, and they are labeled [2] and [1]; thus, there are flow dependences [2] $\longrightarrow_f$ [3] and [1] $\longrightarrow_f$ [3]. ■

3.3. A Collecting Semantics

We now define a *collecting semantics* in which a program's meaning $\mathbf{M_C}$ is the set of all *non-error* states that would be generated by $next_{\mathbf{I}}$ relative to an initial store. We then redefine $p \longrightarrow_f q$ in terms of $\mathbf{M_C}$ and prove the new definition equivalent to the definition of Section 3.2.

The collecting semantics is obtained from the instrumented semantics by extending the latter's definition of the transition function to sets of states:

$$next_{\mathbf{C}} : Prog \rightarrow 2^{State_{\mathbf{I}}} \rightarrow 2^{State_{\mathbf{I}}}$$
$$next_{\mathbf{C}}\ (prog, stateset) =$$
$$\{\ next_{\mathbf{I}}(prog, state)\ |\ state \in stateset\ \} - \{\ \perp_{state}\ \}$$

The program meaning function $\mathbf{M_C}$ is defined as

$$\mathbf{M_C} : Prog \rightarrow Store \rightarrow 2^{State_{\mathbf{I}}}$$
$$\mathbf{M_C}\ (prog) = \lambda\, store\ .$$
$$(fix\ \lambda\, stateset\ .$$
$$\{\ (\mathbf{initial}, (\lambda\, loc.\mathbf{initial}, store))\ \}$$
$$\cup\ (next_{\mathbf{C}}\ (prog, stateset)))$$

The relationship between the instrumented and the collecting semantics is captured by the following lemma:

LEMMA: For all $state \in State_{\mathbf{I}}$,
$prog \vdash_{\mathbf{I}} (\mathbf{initial}, (\lambda\, loc.\mathbf{initial}, store)) \longrightarrow^* state$ iff *state* is a member of $\mathbf{M_C}\ (prog, store)$.

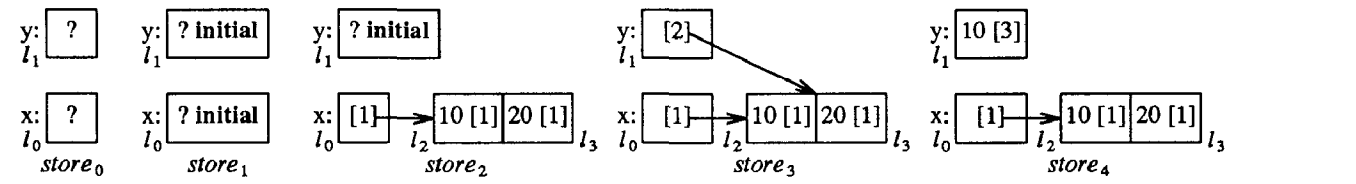

Figure 2. Illustration of the instrumented semantics and of flow dependence under the instrumented semantics. The example program of Figure 1 is repeated with its computation as defined by the instrumented semantics. Flow dependences are determined by examining individual states rather than sequences of states. For example, the flow dependences incident on point [3] are determined by finding all states of the form ([3], $store_i$) and examining the labels in the locations in $store_i$ read by point [3] (namely, the locations that correspond to y and to y.hd). In this example, there is just one such state, the store is $store_3$, the locations are l_1 and l_2, respectively, and they are labeled [2] and [1]; thus, there are flow dependences [2] $\longrightarrow_f$ [3] and [1] $\longrightarrow_f$ [3].

We now define flow dependence relative to the collecting semantics and state a theorem that says that this definition is equivalent to the corresponding definition in the instrumented semantics.

DEFINITION (*Flow Dependence relative to the Collecting Semantics*) : For any program *prog* with program points p and q, there exists a *flow dependence* from p to q iff there exist *store* and state $(q, (label_q, store_q)) \in \mathbf{M_C}(prog, store)$ such that q reads a location labeled p in $store_q$ (relative to *prog* and $label_q$). □

THEOREM 2: For any program *prog* with program points p and q, $p \longrightarrow_f q$ according to the definition of flow dependence relative to the collecting semantics iff $p \longrightarrow_f q$ according to the definition of flow dependence relative to the instrumented semantics.

Example. According to the collecting semantics, the meaning of the program depicted in Figure 2 is the set { (**initial**, $store_0$), ([1], $store_1$), ([2], $store_2$), ([3], $store_3$), (**final**, $store_4$) }. The dependences incident on point [3] can be determined by finding all states of the form ([3], $store_i$) that occur in this set and then examining the labels on the locations in $store_i$ read by point [3]. ■

3.4. An Abstract Semantics

Since the three semantics that we have described so far permit programs to generate arbitrarily many distinct states, none of these semantics yields an effective procedure for computing flow dependences. In this section, we describe a fourth, *abstract* semantics whose meaning function, $\mathbf{M_A}$, yields finite approximations to the (possibly infinite) sets of states produced by $\mathbf{M_C}$. Because $\mathbf{M_A}$ returns finite values, the abstract semantics's characterization of flow dependence is effectively computable.

The meaning of a program in the abstract semantics is given by the program meaning function $\mathbf{M_A}$:

$next_A : Prog \rightarrow 2^{State_A} \rightarrow 2^{State_A}$
$next_A(prog, stateset) =$
  forall $state \in stateset$
    forall $(p', ls') \in nextStateset(prog, state)$
      $\cup$ { $(p', condense(ls'))$ }

$\mathbf{M_A} : Prog \rightarrow Store_A \rightarrow 2^{State_A}$
$\mathbf{M_A}(prog) = \lambda store$.
  (fix $\lambda stateset$.
    $collapse$ (
      { (**initial**, $condense((\lambda loc$. { **initial** }, $store))$) }
      $\cup$ $next_A(prog, stateset)$))

The function *nextStateset*, which maps an abstract state to the set of possible successor states, is analogous to $next_I$. The *condense* operation on labeled stores, and the *collapse* operation on sets of labeled stores are both described below.

Three approximation techniques limit the number of distinct statesets that programs can generate relative to the abstract semantics:

1. The use of a *one-element domain* of atoms prevents statesets from differing only on the value of an atom.
2. A *condense* operation limits store size. Condensation requires that stores map locations to *sets* of values rather than *single* values.
3. A *collapse* operation uses an equivalence relation to limit the size of a set of states.

In the abstract semantics all atoms are represented by the approximate atom o. Without this approximation, the number of stores associated with a statement like "**while** *cond* **do** $x := x + 1$ **od**" would be unbounded, and potentially infinite.

Condensation reduces a store's size by replacing a nonempty region of the store R with a new, single vertex v. Without this approximation, both the number of stores and the sizes of the stores associated with a statement like "**while** *cond* **do** y := cons(10, y) **od**" would be unbounded, and potentially infinite. Vertex v is called a *summary vertex*, since v summarizes the contents of all vertices in R. Let *sel* denote a member of { hd, tl }; then R is condensed to v as follows:

1. Every edge $v_1 \xrightarrow{sel} v_2$, where v_1 is in R, is replaced by $v \xrightarrow{sel} v_2$;
2. Every edge $v_1 \xrightarrow{sel} v_2$, where v_2 is in R, is replaced by $v_1 \xrightarrow{sel} v$;
3. v is labeled o if any vertex in R is labeled o;
4. v's program-point label is the union of all program-point labels on all vertices in R.

Condensation yields a new store that preserves the image of every path in the original store, together with any labels and values that occur along that path.

The abstract semantics uses condensation to maintain the invariant that all store graphs are *k-limited* [12]. By definition, a store graph is k-limited iff every vertex is reachable from a vertex that corresponds to a program variable via a *summary-vertex-free* path of length $\leq k$.

Example. Figure 3 illustrates a use of condensation to limit store size. The region within the dashed box in the left-hand store (*i.e.*, locations l_1-l_4) has been condensed to l_7 in the right-hand store. Note that more than one value is associated with location l_7; the two values associated with l_7 are an atom (o) and a reference to l_7 itself.

The unlabeled edge from l_7 to itself should be regarded as the pair of labeled edges $l_7 \xrightarrow{hd} l_7$ and $l_7 \xrightarrow{tl} l_7$. ■

The third and final approximation allows a smaller set of states to subsume a larger set of states whenever the smaller set of states would yield the same data dependences. Without such an approximation, the number of stores associated with a statement like "**while** *cond* **do** y := cons(10, y) **od**" would be unbounded even after condensation, since a *new* summary vertex v is added to a store whenever a region is condensed.

The collapse operation removes *state* from *stateset* whenever a second *state'* in *stateset* subsumes *state* for the purpose of dependence computation. The collapse operation is defined using a labeled store subsumption relation, $\preceq_{ls}$. We say that *store* is subsumed by *store'*, written $store \preceq_{ls} store'$, if any combination of the following six operations can transform *store* into *store'*:

1. Changing the name of a location that is not associated with an identifier.
2. Adding a program point to a location's label.
3. Adding o to a location's contents.
4. Adding a new, outgoing edge to a summary vertex.
5. Condensing a nonempty set of locations.
6. Inserting a new, empty summary vertex into the store following any non-summary vertex that lacks an outgoing edge.

It is possible to prove that if $store \preceq_{ls} store'$ then *store'* yields a superset of the flow dependences yielded by *store* with respect to *every* program point.

The value of *collapse* (*stateset*) is the value returned by the following procedure:

```
begin
 output := copy (stateset)
 for all (p', store') ∈ stateset
  if (p', store') ∈ output then
   for all (p, store) ∈ output − { (p', store') }
    if p = p' and store ≼ls store' then
     output := output − { (p, store) }
    fi
   endfor
  fi
 endfor
 return(output)
end
```

It is possible to test whether one k-limited store approximates a second in time $O(e \log e)$, where e is the total number of edges in the two stores. It is therefore possible to collapse an n-element *stateset* in time $O(n^2 e \log e)$, where e is the largest number of edges in any pair of stores in *stateset*.

Example. Figure 4 illustrates the use of the *collapse* operation to limit the size of a stateset. Stores $store_1$ through $store_4$ represent four labeled stores that can reach point [2] in the example program. The set $\{ ([2], store_1), ([2], store_2), ([2], store_3) \}$ can be collapsed to $\{ ([2], store_1), ([2], store_3) \}$ for the purpose of dependence computation. Procedure *collapse* eliminates state $([2], store_2)$ from the set because $store_2$ can be transformed to $store_3$ by:

1. condensing the region containing l_1 and l_2 to l_7;
2. adding program point [3] to l_7's label;
3. adding a self-edge to l_7;
4. renaming l_3 to l_5 and l_4 to l_6.

■

Flow dependence is defined relative to the abstract semantics as follows:

DEFINITION (*Flow Dependence relative to the Abstract Semantics*): Let *prog* be any program containing points p and q. There exists a *flow dependence* from p to q iff there exist *store* and state $(q, (label_q, store_q)) \in \mathbf{M}_A(prog, store)$ such that q reads

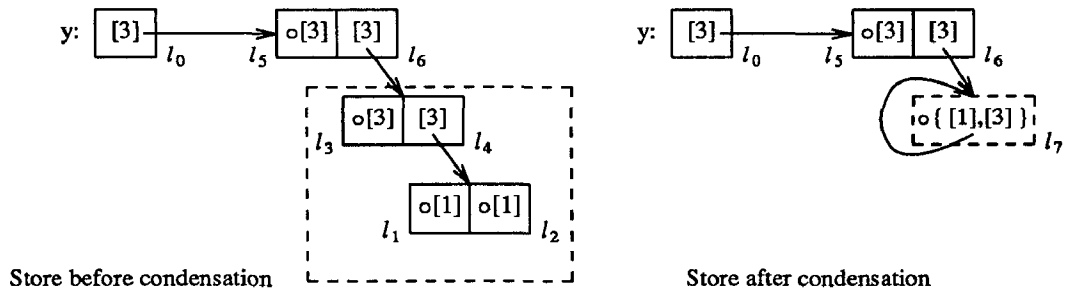

Figure 3. Illustration of the use of condensation to limit store size. The region within the dashed box in the left-hand store has been condensed to form l_7 in the right-hand store.

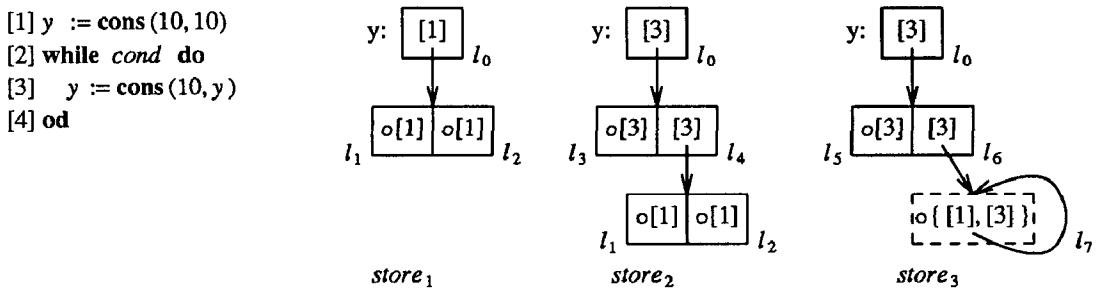

Figure 4. Illustration of the use of the collapse operation to limit stateset size. The set { ([2], $store_1$), ([2], $store_2$), ([2], $store_3$) } can be collapsed to { ([2], $store_1$), ([2], $store_3$) } for the purpose of dependence computation.

a location labeled p in $store_q$ (relative to *prog*, $label_q$, and *nextStateset*). □

The abstract semantics's definition of flow dependence is stated in terms of the sets of abstract states that are generated by $\mathbf{M_A}$ starting from an *abstract* store. To relate the abstract semantics's definition of flow dependence to the collecting semantics's definition of flow dependence, we define a function $abstractStore : Store \rightarrow Store_A$ that maps stores to abstract stores by converting all atoms to o and by condensing all non-k-limited portions of the store.

THEOREM 3: For any program *prog* with program points p and q, $p \rightarrow_f q$ with respect to $store \in Store$ according to the definition of flow dependence in the collecting semantics only if $p \rightarrow_f q$ with respect to $abstractStore(store)$ according to the definition of flow dependence in the abstract semantics.

It is possible to show that the abstract semantics's characterization of flow dependence is safe by proving two assertions about the abstract semantics. The first is that the abstract semantics is an abstract interpretation [4] of the collecting semantics. The second assertion states that if a set of states in the abstract semantics $stateset'_A$ subsumes $stateset_A$, then $stateset'_A$ generates a superset of the flow dependences generated by $stateset_A$ (relative to the abstract semantics's characterization of flow dependence). A proof appears in [10].

3.5. An Algorithm for Computing Flow Dependences

Our algorithm for determining flow dependences yields a safe approximation to a program's flow dependences with respect to a specified environment and initial store. This algorithm is divided into a *reaching-stores phase* and an *inference phase*:

- The reaching-stores phase computes a program's meaning relative to $\mathbf{M_A}$ — thereby associating every program point q with a set of store graphs that approximate the set of memory "layouts" that can arise at q during program execution.
- The inference phase examines the set of stores that reach every program point q. It determines that $p \rightarrow_f q$ iff q reads a location labeled p in a store graph that reaches q.

Taken together, Theorems 1, 2, and 3 (*cf.* Sections 3.2, 3.3, and 3.4, respectively) demonstrate the correctness of this algorithm.

By applying $\mathbf{M_A}$ to a store that approximates all other stores, one can compute a safe approximation to a program's flow dependences relative to any initial store. For a given environment *env*, any store $\top_{store(env)}$ that meets all of the following conditions approximates all other stores:

(1) $\exists$ location *loc* that does *not* correspond to a variable, such that
$\top_{store(env)}(loc) = \{(loc, loc), o\}$
Location *loc* must be a *summary* location.

(2) $\forall$ locations *loc'* that *do* correspond to variables,
$\top_{store(env)}(loc') = \{(loc, loc), o\}$
These locations may *not* be summary locations.

(3) $\forall$ locations *loc''* not covered by (1) or (2) above,
$\top_{store(env)}(loc'') = UNUSED$

Note that the set of stores defined by conditions 1 through 3 differ only on which location corresponds to *loc*. A picture of $\top_{store(env)}$ for the environment that maps x to l_0, y to l_1, and z to l_2 is given in Figure 5.

4. OUTPUT AND ANTI-DEPENDENCES

Our informal definition of an output dependence from program point p to program point q was that p writes a value to a location that is later overwritten by q. Our informal definition of an anti-dependence from program point p to program point q was that p reads a value from a location that is later overwritten by q. The definitions of output and anti-dependence can be formalized relative to the standard semantics, as follows:

DEFINITION (*Output dependence relative to the standard semantics*): For any *prog* with program points p and q, there is an *output dependence* from p to q iff there exists a *store* such that:

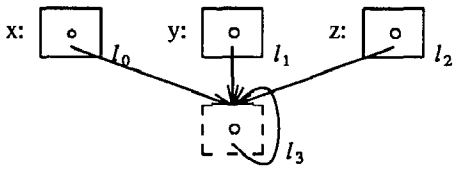

Figure 5. $\top_{store(env)}$, an approximation to all other stores with respect to the environment that maps x, y, and z to l_0, l_1, and l_2, respectively.

$prog \vdash (\textbf{initial}, store) \rightarrow^* (p, store_p) \rightarrow^+ (q, store_q)$;
$loc \in writeset(prog, (p, store_p))$;
for all $(point, store)$ between $(p, store_p)$ and $(q, store_q)$,
$loc \notin writeset(prog, (point, store))$;
$loc \in writeset(prog, (q, store_q))$. □

DEFINITION (*Anti-dependence relative to the standard semantics*): For any *prog* with program points p and q, there is an *anti-dependence* from p to q iff there exists a *store* such that:

$prog \vdash (\textbf{initial}, store) \rightarrow^* (p, store_p) \rightarrow^+ (q, store_q)$;
$loc \in readset(prog, (p, store_p))$;
for all $(point, store)$ between $(p, store_p)$ and $(q, store_q)$,
$loc \notin writeset(prog, (point, store))$;
$loc \in writeset(prog, (q, store_q))$. □

Algorithms similar to the one described in Section 3.5 can be used to determine a safe approximation to a program's output dependences and anti-dependences. The reaching-stores phases of the output and anti-dependence algorithms associate every program point q with a set of store graphs that approximate the set of memory "layouts" that can arise at q during program execution. Every store location v is labeled with a set of program points that describes the location's access history:

- The output dependence algorithm labels *loc* with points that might have *written loc*'s current value. The output dependence algorithm — like the flow dependence algorithm — monitors writes because output dependences arise through a sequence of states that begins with a write.
- The anti-dependence algorithm labels *loc* with points that might have *read loc*'s current value. This algorithm monitors reads because anti-dependences arise through a sequence of states that begins with a read.

The reaching-stores phase of the output dependence algorithm can use the program meaning function $\mathbf{M_A}$ of the abstract semantics as given in Section 3.4. To determine antidependences, we use a different, but related abstract semantics. The abstract semantics for computing anti-dependences is developed from an instrumented semantics that labels every location with all points that might have *read* its current value.

5. DEPENDENCES AND NATURAL LOOPS

A number of different types of interactions between dependences and loops have been identified. These interactions are important in the contexts of automatic parallelization and program integration [2,8,22]. One type of interaction is whether a loop *carries* a dependence, *i.e.*, whether an execution sequence that gives rise to $p \rightarrow q$ can span more than one iteration of a particular loop. A second type of interaction is the *distance* of a dependence with respect to a loop, *i.e.*, the number of

loop iterations that an execution sequence that gives rise to $p \longrightarrow q$ can span.

Our methods for determining a program's data dependences can be extended to classify a dependence's interactions with respect to any *natural loop* L. A *natural loop* [1] is a maximal collection of program points in a control flow graph such that

1. the points are strongly connected;
2. the collection has a unique *entry* point; *i.e.*, all paths from program points outside of the loop to points inside the loop pass through *entry*.

The methods given below label every location *loc* with an auxiliary label that characterizes how the current value of *loc* has propagated through the control flow graph. Location *loc*'s auxiliary label is updated whenever *loc* is propagated across a *backedge* — an edge from a point within loop L to L's entry point. Conservative labeling strategies are used to guarantee that characterizations of loop-dependence interactions are safe.

5.1. Carrying Loops

A dependence $p \longrightarrow q$ is *carried by* loop L if p and q are in L, and one of the execution sequences that gives rise to $p \longrightarrow q$ spans more than one iteration of L. A dependence $p \longrightarrow q$ is *independent of* L if either p or q is outside of L, or if one of the executions sequences that gives rise to $p \longrightarrow q$ is confined to a single iteration of L. Note that there may be both a loop-carried *and* a loop-independent dependence between a pair of program points.

In this section we describe a method for determining whether a program's dependences are carried by loop L. Our method is *safe*; that is,

- the loop-carried dependences determined by the algorithm are a superset of the actual loop-carried dependences;
- the loop-independent dependences determined by the algorithm are a superset of the actual loop-independent dependences.

Our method assigns every location *loc* an additional *status* label that records whether *loc*'s current contents might have propagated across a backedge to L's entry point. The following rules describe how the reaching-stores phases of our algorithms manipulate a location's status label:

1. All locations are initially labeled { **independent** }.
2. Whenever a non-summary location is either initialized or modified, its status label is set to { **independent** }.
3. Whenever a region R is condensed, the new summary location's status label is the union of all status labels on all locations in R.
4. Whenever a summary location is modified, its status label *status* is set to *status* $\cup$ { **independent** }.
5. Whenever information is propagated across a backedge to L's entry point, *every* location's status label is set to { **carried** }.

The inference phase of our method uses status labels to classify dependences. If q reads a location labeled (p, *status*), where p and q are in L and **carried** $\in$ *status*, then the flow dependence algorithm determines that there is a dependence $p \longrightarrow_f q$ that is carried by L. If q reads a location labeled (p, *status*), where either p or q is not in L or **independent** $\in$ *status*, then the algorithm determines that there is a dependence $p \longrightarrow_f q$ that is independent of L. The inference phases of our output and anti-dependence algorithms examine the labels of locations *modified* at q in a similar manner.

This technique can be extended to analyze multiple loops by treating a status label as a function from a set of loops to $2^{\{\textbf{carried}, \textbf{independent}\}}$ and modifying the reaching-stores computation accordingly.

5.2. Dependence Distance

Assume that L is a loop that contains points p and q, and let q be dependent on p. The *distance* of the dependence $p \longrightarrow q$ is the minimum number of times that L must iterate in order to give rise to $p \longrightarrow q$; *i.e.*, the minimum number of backedges to L's entry point that must be traversed in every sequence of states that gives rise to $p \longrightarrow q$. Knowing that the memory accesses that give rise to $p \longrightarrow q$ always span at least n iterations of L may make it possible for a parallelizing compiler to schedule n iterations of a loop in parallel.

In this section we describe a method for determining a dependence's distance with respect to loop L. Our method is safe; that is, it gives an estimate of a dependence's distance that is no greater than the dependence's actual distance. Our method assigns every location *loc* an additional *distance* label that counts how many times *loc*'s current contents might have propagated across a backedge to L's entry point. These distance labels are values in the range from 0 to n, where n is a user-selected bound. The following rules describe how the reaching-stores phases of our algorithms manipulate a location's distance label:

1. All locations are initially labeled 0.
2. Whenever a location is either initialized or modified, its distance is set to 0.
3. Whenever a region R is condensed, the new summary location's distance label is the minimum of all distance labels on all locations in R.
4. Whenever information is propagated across a backedge to L's entry point, *every* location's distance label is incremented (labels that are already n excepted).

The inference phase of our flow dependence algorithm concludes that d is a safe lower bound on the distance of

$p \rightarrow_f q$ if, for every location labeled (p, m) that q reads, $d \leq m$. The inference phases of our output and anti-dependence algorithms apply the same criterion to locations *modified* at q.

This technique can be extended to analyze multiple loops by treating a distance label as a function from a set of loops to $0 \cdots n$ and modifying the reaching-stores computation accordingly.

6. RELATION TO PREVIOUS WORK

The starting point of this paper is the method of Jones and Muchnick for determining an approximation to the "lay-outs" of memory that can possibly arise at each program point during execution [12]. To compute data dependences between constructs that manipulate and access heap-allocated storage, we extended the domain employed in the Jones-Muchnick abstract interpretation so that the (abstract) memory locations are labeled by the program points that set their contents. Flow dependences are then determined from memory layouts according to the component labels found along the access paths that must be traversed during execution to evaluate the program's statements and predicates.

Our method for computing flow dependences is presented as an abstract interpretation. To ensure that our analysis technique applies to *all* implementations, we proved that, for the purpose of determining flow dependences, the standard semantics and the instrumented semantics are equivalent. This result guarantees that the extra information introduced in the instrumented semantics does not embody any implicit implementation commitments. This is in contrast to abstract-interpretation studies that use what has been called "sticky" abstract interpretation, in which the semantic domains of a language's *denotational* definition get extended with "occurrences" or "program points"[5]; in these studies the question of whether the extended definitions introduce implementation commitments is not addressed. In fact, we have not seen *any* previous studies that start from a non-standard semantics (using either denotational or operational semantics) that address the question of whether the non-standard semantics establishes an implementation commitment.

The work reported by Guarna in [7] describes a method for determining a program's data dependences, including loop-carried dependences, in a subset of the C programming language. Guarna's technique fails to account for either circular structures or heap-allocated storage.

Two studies by Larus and Hilfinger concern problems that are related to, but differ from, the dependence computation problem posed at the outset of this report. [15] describes a method for parallelizing recursive invocations of a single procedure that access a common shared object. [16] describes a technique, called conflict analysis, that, like dependence analysis, can be used to determine whether two statements can be executed in parallel. However, conflict analysis is less precise than dependence analysis. As defined in [16], two statements p and q conflict if they both potentially access the same memory location *loc* and at least one of them writes into *loc*, *regardless of whether or not there is an intervening write into loc along the execution path between p and q*. Therefore, a conflict does not necessarily correspond to a dependence. For example, a read/write conflict could actually be a flow dependence (a write-before-read dependence), an anti-dependence (a read-before-write dependence), or no dependence at all.[2]

One consequence of using conflict information rather than dependence information is illustrated by the following example program:

[1] $x := 10$
[2] $x := 20$
[3] $y := x$

(To simplify the example only scalar variables are used and no storage allocation is performed; the problem described below also arises for pointer variables and heap-allocated storage.) The example program contains two read/write conflicts, $[1] \infty [3]$ and $[2] \infty [3]$, but just one flow dependence, $[2] \rightarrow_f [3]$. Now consider how this information would be used to take *program slices*. (The slice of a program with respect to program point p consists of all statements and predicates of the program that might affect the value of variables used at point p [26][3].) The slice of a straight-line program with respect to a point p can be determined by finding all elements on which p has a transitive flow dependence.[4] Using dependence information we find that the slice with respect to statement [3] is:

[2] $x := 20$
[3] $y := x$

In contrast, we must settle for a less precise slice if we are given only conflict information. The slice would be found by treating the two read/write conflicts, $[1] \infty [3]$ and $[2] \infty [3]$, as the flow dependences $[1] \rightarrow_f [3]$ and $[2] \rightarrow_f [3]$, yielding the slice

---

[2] The loop-unrolling technique for distinguishing between flow and anti-dependences, described in [17], does not work for all examples.

[3] Program slicing can be used to isolate individual computation threads within a program, which can help a programmer understand complicated code. Program slicing is also used by the algorithm for automatically integrating program variants described in [8].

[4] Dependence information can also be used to find slices of programs with loops, conditional statements, and procedure calls [9,21,23]. For example, in a program with loops and conditional statements, the slice with respect to point p is determined by finding all elements on which p has a transitive flow or *control* dependence.

[1] $x := 10$
[2] $x := 20$
[3] $y := x$

This slice contains statement [1], although the assignment to x at [1] cannot affect the value of x used at [3].

After reading a previous version of this paper, Larus extended his techniques to perform dependence analysis in addition to conflict analysis [18, 19]. The dependence-analysis technique reported in [18] combines a standard dataflow technique for computing reaching definitions [1] with the alias-analysis technique described in [16]. [18] also describes methods for determining a dependence's carrying loops and its distance relative to an enclosing loop, and gives a method for computing *def-order* dependences [8].

A major difference between our dependence-analysis technique and [18] is that we provide a proof of correctness that relates the proposed static analysis to the language's semantics. A second difference between the two techniques concerns the representation of memory. The alias-analysis technique from [16] computes a *single* structure at each program point p that depicts all possible aliases that could arise at p. The use of a single structure at every program point to represent aliases saves space but also yields less precise analyses, particularly in the context of programs that add new locations to the head of existing structures.

Other work on the analysis of languages with heap-allocated storage does not address the computation of data dependences. This includes [3] and [24], which give techniques for determining which program points may have *allocated* the structures pointed to by a program's variables (as opposed to which program points may have most recently written into those structures, as in our work), [11], which concerns how to determine lower bounds on reference counts, [20], which describes a method for analyzing side effects and aliasing in higher-order functional languages, and [25], which describes a lattice for data-flow analysis of languages with heap-allocated storage.

7. ACKNOWLEDGEMENTS

We thank Marvin Solomon for bringing [12] to our attention, and Will Winsborough for his help in developing the meaning functions for the standard and collecting semantics.

REFERENCES

1. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

2. Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Dept. of Math. Sciences, Rice Univ., Houston, TX (April 1983).

3. Chase, D.R., "Garbage collection and other optimizations," Ph.D. dissertation, Dept. of Computer Science, Rice Univ., Houston, TX (August 1987).

4. Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238-252 in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, (Los Angeles, CA, January 17-19, 1977), ACM, New York, NY (1977).

5. Donzeau-Gouge, V., "Denotational definition of properties of program computations," in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

6. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* 9(3) pp. 319-349 (July 1987).

7. Guarna, V.A. Jr., "A technique for analyzing pointer and structure references in parallel restructuring compilers," CSRD Tech. Rep. 721, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL (January 1988).

8. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

9. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7)(July 1988).

10. Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," <In preparation>, Computer Sciences Department, University of Wisconsin, Madison, WI (1989).

11. Hudak, P., "A semantic model of reference counting and its abstraction," pp. 45-62 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky and C. Hankin,Ellis Horwood Limited, Chichester, West Sussex, England (1987).

12. Jones, N.D. and Muchnick, S.S., "Flow analysis and optimization of Lisp-like structures," in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

13. Kennedy, K., "A survey of data flow analysis techniques," in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

14. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., "Dependence graphs and compiler optimizations," pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

15. Larus, J.R., "Curare: Restructuring Lisp programs for concurrent execution," UCB/CSD 87/344, Computer Science Division, Dept. of Elec. Eng. and Comp. Sci., Univ. of California – Berkeley, Berkeley, CA (February 1987).

16. Larus, J.R. and Hilfinger, P.N., "Detecting conflicts between structure accesses," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 21-34

(July 1988).

17. Larus, J.R. and Hilfinger, P.N., "Restructuring Lisp programs for concurrent execution," *Proceedings of the ACM/SIGPLAN PPEALS 88* (New Haven, CT, July 19-21, 1988), *ACM SIGPLAN Notices* **23**(9) pp. 100-110 (September 1988).

18. Larus, J.R., "Refining and Classifying Data Dependences," unpublished extended abstract, Berkeley, CA (November 1988).

19. Larus, J.R., *private communication.* 1988.

20. Neirynck, A., "Static Analysis of Aliases and Side Effects in Higher-Order Languages," Ph.D. dissertation, Computer Science Department, Cornell University, Ithaca, NY (February 1988).

21. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

22. Padua, D.A., "Multiprocessors: Discussion of some theoretical and practical problems," Ph.D. dissertation and Tech. Rep. R-79-990, Dept. of Computer Science, University of Illinois, Urbana, IL (November 1979).

23. Reps, T. and Yang, W., "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

24. Ruggieri, C. and Murtagh, T.P., "Lifetime analysis of dynamically allocated objects," pp. 285-293 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

25. Stransky, J., "Analyse sémantique de structures de données dynamiques avec application au cas particulier de langages LISPiens," Ph.D. dissertation, Université de Paris-Sud, Centre d'Orsay (June 1988).

26. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).