DEPENDENCE-BASED REPRESENTATIONS FOR PROGRAMS WITH REFERENCE VARIABLES

by

Phillip E. Pfeiffer, IV

Computer Sciences Technical Report #1037

August 1991

DEPENDENCE-BASED REPRESENTATIONS FOR PROGRAMS WITH REFERENCE VARIABLES

by

PHILLIP E. PFEIFFER, IV

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Science)

at the
UNIVERSITY OF WISCONSIN—MADISON
1991

Abstract

Three features common to modern programming languages are popular because they simplify the development of efficient programs. The first, the *assignment statement*, allows the components of a data structure to be redefined as a computation progresses. The second, *dynamic allocation*, allows memory for data structures to be acquired, destroyed, and reused at need. The third, the *reference* (*i.e.*, *pointer*) *variable*, allows multiple data structures to share a common substructure. These three features, unfortunately, make it difficult to estimate program behavior at compile-time. Such estimates play a crucial role in the (automatic) improvement, modification, and reuse of existing software.

The first part of this thesis develops a family of algorithms that characterize a program's data dependences, with respect to an example structured language with assignment statements, reference variables, dynamic allocation, and procedures. Intuitively, a data dependence $p \longrightarrow_d q$ asserts that a statement q manipulates a data object that was first manipulated by p. The analyses developed here estimate a program's data dependences, with respect to the example language's *implementation semantics* and an arbitrary set of initial stores. (This claim is established with the aid of *Abstract Interpretation*, a formalism for showing such analyses correct.) These algorithms are also flexible: they return a safe estimate of a program's dependences with respect to several common strategies for estimating program behavior (*i.e.*, for using a bounded set of approximate states to estimate the unbounded set of states that a program might generate). These strategies for estimating program behavior are surveyed and critiqued, and extensions to one of these techniques, known as *k-limiting*, are proposed.

The second part of this thesis concerns the safety of using dependences to reason about program behavior. Earlier authors have shown that specific types of *dependence-based representations* (*i.e.*, dependence-depicting graphs) model specific facts about program execution. None of these results, however, apply to languages with reference variables, dynamic allocation, and procedures. This thesis proves that pointer-language programs that have isomorphic dependence-based representations are behaviorally equivalent.

Table of Contents

1. INTRODUCTION

The proof of a system's value is in its existence. —A. Perlis [Per82]

Classic imperative programming languages have important weaknesses—limitations and missing features that make programs unnecessarily difficult to develop and understand. Some of these weaknesses are described in an essay by John Backus [Bac78]. In his 1977 Turing Award lecture, Backus argues that a good programming language should provide high-level operations on aggregate types. Backus observes that high-level operators like Lisp's *map* functionals allow users to develop succinct and clear statements of algorithms. Backus also argues that languages should minimize the use of operators with side effects, since these operators make it more difficult to assess a computation's net effect. Conventional, imperative languages like FORTRAN are criticized for making an operator that violates both precepts, the scalar assignment statement, the basis of all computation.

Some authorities believe that these classic programming languages should simply be abandoned. Backus, for example, proposed that future research into programming languages be limited to what are now called *declarative languages*. A declarative language is a language that in theory allows a programmer to specify *what* a program computes, without having to specify a computation's intermediate steps. It is worth noting that declarative paradigms such as Backus's functional programming [Bac78] have evolved to where they are sometimes credible alternatives to imperative languages (*cf.* [Hud88]). Proponents of eliminating imperative languages, however, usually ignore the negative consequences of abandoning the user community's massive investment in imperative languages and software.

Other, more conservative approaches have been developed for coping with the flaws of imperative languages. One such approach emphasizes the use of utilities that make programs easier to manipulate. These utilities use information about a language's semantics to make judgments about how programs evaluate. Examples of such utilities include

* *optimizing compilers* [Kuc81], which automatically improve program efficiency;
* *parallelizing compilers* [Kuc81], which automatically transform programs into parallel programs;
* *program-comparison utilities* [Hor90], which use *semantic* criteria to identify differences between variants of a program; [1]
* *program-integration utilities* [Hor89], which merge multiple versions of a base program *b* into a single program that preserves all non-conflicting changes to *b*; and
* *flowback debuggers* [Mil88], which use information about a program's form to minimize the size of a stored trace.

A crucial component of many such utilities is a ***dependence-based representation*** of a program's semantics. A *dependence* is an assertion that one statement might affect how a second statement evaluates. A *dependence-based representation* (***dbr***) is a graph that summarizes a program's dependences. A *dbr* for a

[1] Semantic program comparators can be distinguished from text-based comparators like *diff* [Hun76], which, lacking information about a program's meaning, make imprecise—and erroneous—judgments about how programs differ (*cf.* [Hor89]).

program P consists of a set of vertices that represent P's statements, and a set of labeled edges that represent possible dependences between a program's statements. An example *dbr* is depicted in Figure 1.1.

Dbrs are popular because they simplify program analysis. A *dbr*, once constructed, gives a useful estimate of a program's threads of computation—one that is reasonably precise and easily manipulated. One can, for example, determine whether a statement p might affect the evaluation of statement q by determining whether a program's *dbr* contains a path from p to q. In this manner, *dbrs* expose a program's potential parallelism, and simplify the task of assessing program behavior.

Dbrs have three additional characteristics that make them well-suited to program analysis:

- *Dbrs* are flexible. The kinds of dependences that are represented in a *dbr* can be adjusted according to that *dbr*'s intended use.
- *Dbrs* are tunable. An algorithm for computing a program's dependences can be adjusted according to an analysis's available resources. Coarse (but fast) analyses yield rough but safe estimates of a program's dependences. Slower analyses yield *dbrs* that are no worse—and probably better—estimates of program evaluation. *Dbrs* can also be tuned by hand. Utilities such as PTOOL [All86] and Curare [Lar89] allow their users to specify that certain dependences cannot arise.
- The semantics of certain kinds of *dbrs* are well understood. For several example languages, it has been proven that *dbrs* provide a sound characterization of a program's meaning,

From the late 1960's through the mid 1980's *dbrs* were used primarily for parallelizing FORTRAN programs. An early paper by Ramamoorthy and Gonzalez [Gon69] introduces graph-based techniques for

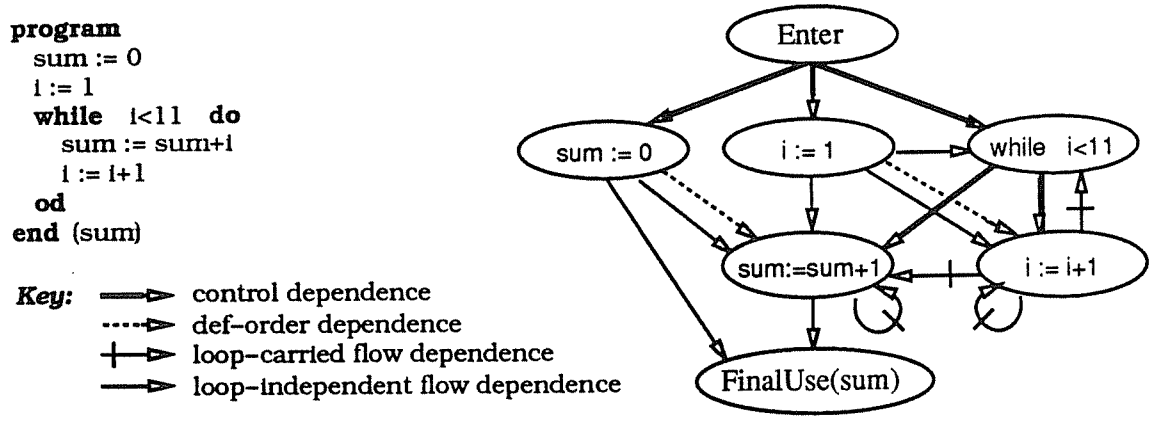


Figure 1.1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable SUM. The figure also shows a type of dependence-based representation for this program known as a *pdg*. (Dependences are defined in Chapter 3, and *pdgs* in Chapter 7.)

parallelizing FORTRAN. Kuck, Muraoka, and Chen, who cite the Ramamoorthy paper, were among the first to recognize the importance of distinguishing among different types of data dependence [Kuc72]. Kuck *et. al.* later developed many of the fundamental *dbr*-based algorithms for restructuring FORTRAN [Pad79, Kuc81]. Other important contributions that were made during the initial period of *dbr* research include work by researchers at Rice University (*e.g.*, [All84]), Michigan Tech (*e.g.*, [Ott84]), and IBM (*e.g.*, [Fer87]).

Current research on *dbrs* addresses other languages, other problems, and other concerns. New kinds of *dbrs* have been developed that support languages with *while* loops and procedures [Hor89, Hor90a]. Others have also been developed for studying specific aspects of program behavior (*e.g.*, [Cyt86, Hor89, Yan90]). Another relatively recent development is the increased emphasis on understanding the semantics of *dbrs* (*cf.* [Rep89, Ram89, Sel89]).

This thesis represents another step in the continuing search for a more general theory of *dbrs*. It is concerned with the computation and characterization of *dbrs* for ***pointer languages***—imperative languages that support heap-allocated storage, reference variables, and assignment statements that overwrite the contents of reference variables. There are four concerns that motivate the work reported in this thesis:

- Heap allocation, reference variables, and assignment statements are useful constructs: they simplify the task of programming with shared and circular structures, and with structures whose size is not known until run-time.
- *Dbrs* for programs that use pointers should provide a reasonable estimate of how such programs might evaluate. Making worst-case assumptions about the use of pointers can yield estimates of program behavior that are too coarse to be useful [Lan90].
- The semantics of pointer-language *dbrs* had not been investigated prior to the inception of this research.
- Pointer languages are a challenging area of research. Destructive pointer updating complicates the analysis of pointer languages by forcing the recomputation of aliases at every assignment statement. Allocation complicates dependence analysis by forcing analyses to construct finite approximations to potentially infinite sets of memory configurations. Finally, proofs about the semantics of *dbrs* must account for how programs manipulate references and allocate storage.

The remaining seven chapters of the thesis can be divided into four parts. Chapter 2 describes an example pointer language. Chapters 3 through 6 develop algorithms for computing a program's dependences w.r.t. this example language. Chapter 7 defines an example *dbr* for this language, and argues that this *dbr* characterizes a program's meaning. Chapter 8 then summarizes the work presented in presented in the preceding five chapters. The following is a detailed synopsis of the presentation.

Part I. An Example Pointer Language

Chapter 2 defines the model language that forms the basis of this study. This language, called language $\mathcal{H}$, supports heap-allocated storage, reference variables, lexically-scoped recursive procedures, and structure declarations. The semantics for language $\mathcal{H}$, $\mathbf{M}_{\mathcal{H}}$, gives an *operational* characterization of a program's meaning.

Part II. Computing Data Dependences for Pointer Programs

Chapters 3 through 6 develop algorithms for computing a pointer program's data dependences. They also prove that the dependences computed by these algorithms are a ***safe estimate*** of (*i.e.*, a superset of) the dependences that a program exhibits w.r.t. $\mathbf{M}_{\mathcal{H}}$.

Chapter 3 defines a program's dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$. The chapter first explains the concept of dependence. Chapter 3 then uses $\mathcal{H}$'s definition to formalize the standard notions of control and data dependence, and to define a refinement of the notion of data dependence. This refinement, the *carriers* of a dependence, characterizes how a data dependence interacts with a program's loops and procedures.

Chapter 4 develops new definitions of data dependence that simplify the task of dependence computation. In Chapter 3, the notion of a data dependence is defined as function of the *sequence of states* that computations generate. In Chapter 4, data dependence is redefined as a function of the *individual states* that computations generate. These new definitions of dependence are based on a second semantics for language $\mathcal{H}$. This semantics, $\mathbf{MI}_{\mathcal{H}}$, is an extension of $\mathbf{M}_{\mathcal{H}}$ that labels objects in stores with values that characterize a computation's history. These labels, for example, identify those statements that might have read or written the various structures and references in a program's stores. Chapter 4 shows that the new definitions of dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ are equivalent to the definitions of dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$ given in Chapter 3.

Chapter 5 uses $\mathbf{MI}_{\mathcal{H}}$ to develop algorithms for estimating a program's dependences. These algorithms first estimate the set of all states that a computation might generate w.r.t. $\mathbf{MI}_{\mathcal{H}}$. This set of states is then used to estimate the set of data dependences that a computation might exhibit w.r.t. $\mathbf{MI}_{\mathcal{H}}$. The semantics that generates these estimates of program behavior, $\mathbf{MA}_{\mathcal{H}}$, is an extension of $\mathbf{MI}_{\mathcal{H}}$ that uses *abstract* (*i.e.*, "approximate") *stores* to obtain a conservative, terminating characterization of a program's behavior. The claim that $\mathbf{MA}_{\mathcal{H}}$ can be used to estimate a program's dependences w.r.t. $\mathbf{MI}_{\mathcal{H}}$ is proved with a theoretical framework known as *Abstract Interpretation* [Cou77]. This assertion that $\mathbf{MA}_{\mathcal{H}}$'s characterization of dependence is safe w.r.t. $\mathbf{MI}_{\mathcal{H}}$'s, when combined with the assertion that $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{M}_{\mathcal{H}}$ give equivalent characterizations of a program's dependences, implies that these algorithms compute a proper superset of a program's dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$.

The definition of $\mathbf{MA}_{\mathcal{H}}$ given in Chapter 5 does not specify *how* termination is to be achieved; it merely assumes that an analysis has been supplied with an operator that, intuitively, limits the number of distinct states that an analysis can return. Chapter 6 explores possible definitions for this operator. The greater part of Chapter 6 is devoted to a comparison of techniques for estimating stores. Chapter 6 also proposes a modified version of a standard store-limiting technique known as k-limiting, and argues that the modified k-limiting operator can be used to obtain space-efficient abstract stores.

Part III. A Dependence-Based Representation for Pointer Programs

Chapter 7 argues that a pointer program's dependences can be used to reason about its meaning. This chapter first defines an example dependence-based representation for language $\mathcal{H}$. This *dbr*, the ***heap-language system dependence graph*** (***hsdg***) is related to an earlier *dbr* for languages with procedures, the *system dependence graph* (*sdg*). Chapter 7 next argues that programs with isomorphic *hsdgs* have equivalent meanings. Chapter 7 then concludes with a critique of this *dbr*, and suggests avenues for further research.

Part IV. Conclusions

Chapter 8 summarizes the work presented in this thesis. It also lists open problems, and discusses possible extensions to the framework developed in Part II.

The author has tried to simplify the reader's task in the following five ways. Chapters 3 through 7 are prefaced with short introductions that sketch these chapters' contributions to the literature. Comparisons with related work are typically relegated to the end of each chapter; the lone exception is Chapter 6, which can be regarded as a gloss on Chapter 5. Semantic definitions and the proofs of certain low-level lemmas have been consigned to appendices. The proof of another crucial result, the Pointer-Language Equivalence Theorem (*cf.* §7.3), is sketched before being given in detail. Finally, there are two special indices at the back of the thesis: the first lists authors, and the second lists definition sites for ***special terms***—technical terms that appear in ***bold italics*** (names of lemmas, section headings, and terms in the concluding chapter excepted).

OTHER REMARKS ABOUT THE THESIS

Backus described two deficiencies of common imperative languages: their reliance on side effects, and their lack of support for aggregate data structures. This thesis is concerned with one specific approach for dealing with the first of these deficiencies. The data structure that is at the heart of this approach, the *dbr*, characterizes one kind of assertion about program behavior—the dependence. Approaches to understanding how programs evaluate that use other facts about program behavior (*e.g.*, [Ger75, Par83]) are beyond the scope of this thesis. Ideas for ameliorating the aggregate problem—*e.g.*, extensions of FORTRAN that support APL-like matrix primitives [Ame89]—are also beyond the scope of the thesis.

2. AN EXAMPLE LANGUAGE WITH DYNAMIC ALLOCATION

Be careful in the beginning, and you have no trouble in the end. —the *I Ching*, cited in [Cle88]

This thesis is concerned with languages that support ***dynamic allocation***. Dynamic allocation is a strategy for managing program memory that sets aside space for structures "on demand"—*i.e.*, when certain operators, called *allocation operators*, are evaluated. These structures persist until they are no longer needed or the computation terminates. Dynamic allocation is one of three common strategies for managing program memory. The other two strategies, *static allocation* and *stack allocation*, differ from dynamic allocation—and from each other—according to how structures are created and destroyed. Each of these strategies has certain advantages. Dynamic allocation simplifies the task of programming with structures whose size and useful lifetime are unknown at compile-time.

Dynamic allocation is typically implemented by providing each program with a pool of auxiliary memory locations known as the *heap*. Allocation operators acquire structures from the heap by

- obtaining unused locations from the heap's set of free locations (*i.e.*, its ***freelist***);
- removing these locations from the freelist; and
- (optionally) tagging the set of newly-acquired locations with a *type* that identifies these locations as a single, logical entity.

Other operators are then used to access and update heap-allocated structures. Some languages (*e.g.*, Pascal and C) also support an explicit ***deallocation operator***—an operator that returns a structure to a freelist (*i.e.*, when its useful lifetime ends). Other languages, such as Lisp and CLU, automatically return structures that can no longer be accessed to the freelist. This process of automatically retrieving inaccessible locations from the store is known as ***garbage collection***.

The paragraphs that follow define an example language, $\mathcal{H}$, that exhibits properties of languages that support dynamic allocation. Language $\mathcal{H}$ is a structured, first-order language that provides user-defined types; non-nested, recursive procedures; statically-scoped variable declarations; an allocation operator; and a pointer-updating assignment statement.

Figure 2.1 defines $\mathcal{H}$'s abstract syntax. A *program* in language $\mathcal{H}$ is a two-part object that consists of an optional set of *structure declarations*, followed by a set of *procedure declarations*. Structure declarations define a program's allocatable structures. Procedure declarations define a program's executable code. An example program is depicted in Figure 2.2.

Appendix 1 gives the formal definition of language $\mathcal{H}$'s meaning function, $\mathbf{M}_{\mathcal{H}}$. Semantics $\mathbf{M}_{\mathcal{H}}$ is an operational, state-transition semantics for language $\mathcal{H}$—the sort of semantics that could be used to implement $\mathcal{H}$. The use of an *operational* semantics allows one to reason about the sequence of intermediate states that a computation generates (§3.2.2).

A program in language $\mathcal{H}$ is *interpreted* as a partial function from an input store to an output store. Intuitively, a *store* is a directed labeled graph that represents memory. More formally, a store is a map that assigns a unique address (*i.e.*, element of *Loc*) to each of a program's *structures*. A *structure* s consists of a type; an atom, which identifies s's value (if s is atomic); and a map m that names s's successors. This map m maps selectors such as *hd* and *tl* to elements of $Loc_{\perp}$. Every selector that a structure of type s does not

Program	→ {*Structs*;} {*Procs*}
Structs	→ *Struct* {; *Struct*}*
Struct	→ **struct** TYPE **is** < *Sels* >
Sels	→ SEL {, SEL}*
Procs	→ *Proc* {; *Proc*}*
Proc	→ *Recproc*
	→ *Stdproc*
Recproc	→ **recursive** *Stdprocs* **endrec**
Stdprocs	→ *Stdproc* {; *Stdproc*}*
Stdproc	→ **procedure** *Stdp* **end**
Stdp	→ *Proc_hd* {*Local*;} *Stmts*
Proc_hd	→ IDENT ({*Idents*})
Local	→ **local** *Idents*
Idents	→ IDENT {, IDENT}*
Stmts	→ *Stmt* {; *Stmt*}*

Stmt	→ **while** *Cond* **do** *Stmts* **od**
	→ **if** *Cond* **then** *Stmts* {**else** *Stmts*} **fi**
	→ *SelExp* := *Exp*
	→ **call** IDENT (*SelExps*)
	→ **return**
Cond	→ **typeOf** (*SelExp*) **is** TYPE
	→ *SelExp* **Eq** *SelExp*
	→ *SelExp* > *SelExp*
	→ *SelExp* $\stackrel{?}{=}$ *SelExp*
	→ *SelExp* < *SelExp*
	→ **not** *Cond*
Exp	→ **new** (TYPE)
	→ PRIMOP({*SimplExps*})
	→ *SimplExp*
SimplExps	→ *SimplExp* {, *SimplExp*}*
SimplExp	→ ATOM | *SelExp*
SelExps	→ *SelExp* {, *SelExp*}*
SelExp	→ IDENT{ . SEL}*

SEL is a set of alphanumeric names. TYPE is a set of alphanumeric (type) names that includes *atom* and *env*. IDENT is a set of alphanumeric (variable and function) names.

PRIMOP is an unspecified set of primitive operations on atoms. PRIMOP includes '+' and other arithmetic operations.

ATOM is a topped and lifted set of primitive objects that includes **nil**.

Figure 2.1. The abstract syntax of language $\mathcal{H}$

support is mapped to $\perp$, the error location.

Language $\mathcal{H}$ defines two types of built-in structures. *Environments* are structures that map identifiers to structures. *Atoms* are structures that map all selectors to $\perp$. Other structures are defined by the evaluation of structure declarations. A structure declaration names a structure type t and specifies the selectors that t supports. For example, the declaration "**struct** *conscell* **is** < *hd*,*tl* >" defines a two-field structure named *conscell* with selectors *hd* and *tl*.

Language $\mathcal{H}$ supports one kind of store-access expression: the ***identifier expression***. An identifier expression is a string of the form $x.sel_1.sel_2.\cdots.sel_n$, where x is an identifier and the sel^* are selectors. The identifier portion of an identifier expression is interpreted according to a *C*-language-like, two-level scoping discipline. Let $x.sel_1.\cdots$ be an identifier expression that occurs in a procedure P. Then x is interpreted relative to P's local environment when x is either (1) one of P's formal parameters, or (2) declared to be local to procedure P. Otherwise, x is interpreted relative to P's global environment. The rest of an identifier expression is interpreted according to the standard rules for dereferencing structures. If x, for example, references the cons cell <1,2>, then $x.hd$ and $x.tl$ reference "1" and "2", respectively.

```
struct conscell is <hd, tl>;
struct intptr is <intp>;

procedure main ()
 local total;
  /**** sum values in list,
  /**** and determine list length
  total := new(intptr);
  call sum(list, total);
  /**** final sum in total.intp,
  /**** length in listlen
end;

procedure sum (list, result)
 result.intp := 0;
 listlen := 0;
 call sumelts (list, result);
end;

recursive
  procedure sumelts (list, sum)
   if list ≠ nil then
    sum.intp := sum.intp + list.hd;
    listlen := listlen + 1;
    call sumelts (list.tl, sum)
   fi
  end
endrec
```

Figure 2.2. An example program in language $\mathcal{H}$. The program sums a list of atoms. Subroutine *sumelts*( ) uses the global identifier *listlen* to record the length of the list. (Undeclared identifiers such as *listlen* are interpreted w.r.t. a program's global environment.)

Language $\mathcal{H}$ provides three kinds of operations on atoms: arithmetic operators such as '+'; predicates such as $<$, $\stackrel{?}{=}$, and $>$; and logical negation ("¬").

Language $\mathcal{H}$ provides four operations on non-atomic structures: ".", **Eq**, **typeOf**, and **new**. The "." (dereference) operator is described above. **Eq** is a binary predicate that tests whether two selector expressions reference the same structure. **TypeOf** identifies the type of the structure referenced by its operand. New(t) returns a reference to a previously unreferenced structure of type t. Function **new** also initializes the structure that it returns. The evaluation of the expression "**new**(t)", where t is the type $< field_1, \cdots, field_n >$, creates $n+1$ new structures: one structure of s_{new} of type t, and n **nil**-valued atoms. Each of the $field_i$'s in the s_{new} returned by "**new**(t)" references a distinct **nil**-valued atom.

A procedure consists of a two-part *header* and a *body*. A procedure's header names its formal parameters and local identifiers. A procedure's body is made up of while loops, conditional statements, assignment statements, procedure calls, and return statements. While loops and conditional statements have their usual meaning. Assignment statements alter references; a statement like x.hd := 0 replaces a reference x.hd with a new reference to the atom 0.

Semantics $\mathbf{M}_{\mathcal{H}}$ uses the following five-step, pass-by-reference discipline to implement a statement like "**call** $A\ (a_1, \cdots, a_n)$":

1. A set of $n+1$ special references are created at the caller's local environment. One of these references identifies (*i.e.*, points to a special structure that contains) the return address for the call to A. The remaining n references point to the structures denoted by a_1 through a_n.
2. Control is transferred to procedure A.
3. Procedure A creates its local environment.
4. Procedure A initializes its local environment. A reference named *_prev* is first created from A's

local environment to the caller's local environment. (The **return** statement uses this reference to restore the caller's local environment.) The $n + 1$ special references created at step (1) are then used to initialize A's local references. One reference, reference *_callctxt*, is set to A's return address. The remaining n references are set to the initial values of A's formal parameters.

5. Control is transferred to the first executable statement in A.

The return statement resets the local environment to the caller's local environment, then returns control to the program point referenced by *_callctxt*. There is an implicit return statement after the last executable

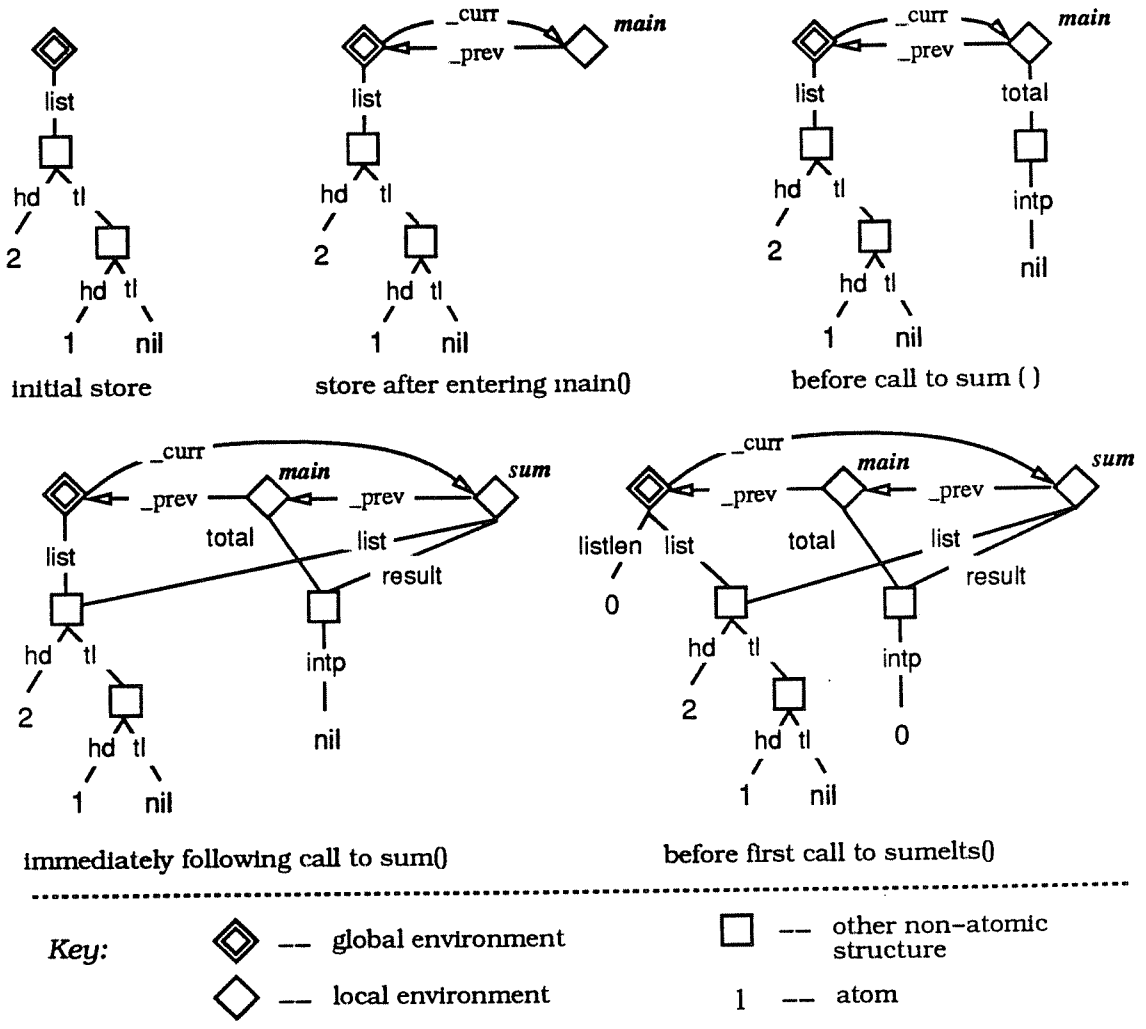


Figure 2.3. The first of two figures illustrating an evaluation of the program in Figure 2.2. Environments are labeled with names of their instantiating procedures; *genv* represents the global environment.

statement in every procedure.

Language $\mathcal{H}$'s meaning function executes a program P by creating and initializing P's freelist, then entering procedure *main* ( ). The following three constraints are imposed on every initial store σ:

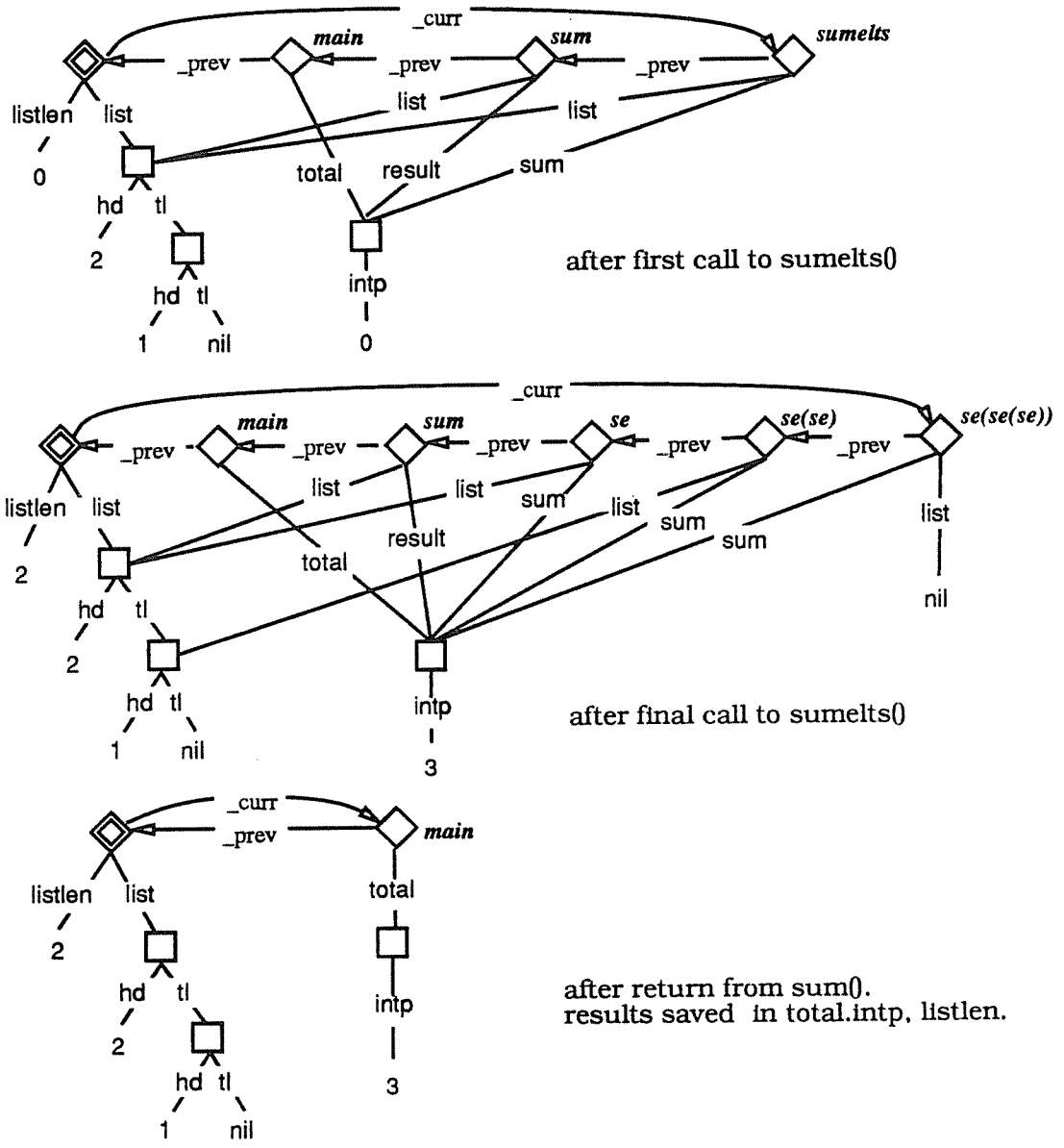


Figure 2.4. The second of two figures illustrating an example computation.

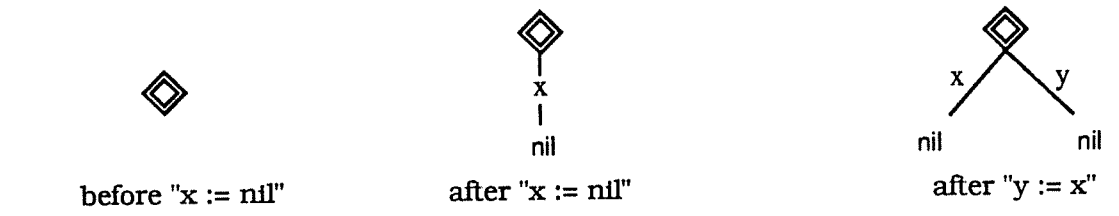


Figure 2.5. Language $\mathcal{H}$'s implementation of atoms. Every structure points to its own copy of an atom.

* σ must be *finite*: σ must contain finitely many accessible structures and references.
* σ must be *deterministic*: every structure may have at most one reference labeled *sel*.
* σ must have exactly one environment. This environment must not be the target of any references.

Figures 2.3 and 2.4 depict the interpretation of the program in Figure 2.2 w.r.t. an initial store that contains a two-element list. These directed labeled graphs depict the store at successive stages of P's evaluation. Nodes represent structures; edges represent references. The special references labeled *_curr* identify the currently active procedure's local environment. The special references labeled *_prev* are used to stack and unstack local environments on procedure entry and exit. Other special references to parameters and return addresses have been omitted for simplicity; a detailed specification of $\mathbf{M}_{\mathcal{H}}$'s calling protocol is given in Appendix 1.

The example computation depicted in Figures 2.3 and 2.4 illustrates three simplifying assumptions that $\mathbf{M}_{\mathcal{H}}$ makes about a program's evaluation. The first such assumption is that **new**() always succeeds. The **new** operation uses a list of inaccessible structures known as the *freelist* to obtain unused structures. A call to **new**() removes the first structure from the freelist and returns a reference to that structure. This thesis makes the assumption that the freelist is unbounded; *i.e.*, that the **new**() operator never fails for want of storage. This assumption makes it possible to sidestep the need for garbage collection. Eliminating garbage collection, in turn, eliminates potential interactions between program points that arise from the reuse of structures: *i.e.*, an allocation operator never operates on a structure that was manipulated by a previous statement.

The second simplifying assumption is that procedure activation records are allocated from the heap, and *not* from a run-time stack of spare locations. Using the heap as the basis for allocation makes $\mathcal{H}$'s definition more uniform. It also eliminates potential interactions between program points that arise from the reuse of the stack: *i.e.*, a call statement never overwrites a location in the stack that was manipulated by a previous statement.

The third simplifying assumption is that operations that return references to atoms return references to unshared atoms. Specifically, an expression like **new**(*conscell*) returns a cons cell whose fields reference two unshared, **nil**-valued atoms. Also, the evaluation of a pair of statements such as "$x :=$ **nil**; $y := x$" yields a store in which x and y reference distinct **nil**-valued atoms (*cf.* Figure 2.5). This assumption simplifies the task of comparing the contents of one store to the contents of a second: such maps play a key role in the development of a non-standard, approximation semantics for language $\mathcal{H}$ (*cf.* Chapter 5).

Section 7.4, which discusses these three simplifications in more detail, argues that none of these assumptions has a major effect on the analyses developed in this thesis.

Language $\mathcal{H}$ has also been simplified by the omission of several common programming constructs. These include:

- *Support for arbitrary scoping.* The two-level scoping mechanism assumed in this chapter is a compromise between simplicity and generality. Supporting no more than two levels of scoping simplifies $\mathcal{H}$'s definition. Supporting more than one level of scoping makes the presentation more realistic, and suggests how these results can be extended to languages with more general scoping.
- *Value-returning procedures.* The effect of a value-returning procedure call such as "*value* := **call** *sumfn* (*list*)" can be obtained by defining *sumfn* as a two-parameter function, and setting the second parameter to reference the result (*cf.* Figure 2.2).
- *Static typing.* Language $\mathcal{H}$'s run-time type discipline is similar to Lisp's. The effort required to define a stronger typing system would have detracted from the focus of the presentation.
- *Input and output, arrays, higher-order procedures, gotos, dynamic scoping, and reference arithmetic.* These features are beyond the scope of this thesis. How the addition of these features to the example language would have complicated the presentation is considered in Chapter 8.

OTHER REMARKS ABOUT CHAPTER 2

Static, stack, and dynamic allocation are discussed in more detail in the opening chapter of Ruggieri's thesis [Rug87]. This well-written overview of storage management also discusses *persistent allocation*—a fourth allocation strategy that preserves structures across computations (*i.e.*, on backing store).

Formalisms that have been used to model heaps include location-based models of memory [Ple81, Deu90]; collections of symbolic equations [Sch75]; path strings [Myc81, Ino88]; connection matrices of path strings [Hen89, Hen90]; and alias sets [Har89, Gua90]. The formalism used in this thesis borrows heavily from Jones's and Muchnick's original scheme for depicting stores [Jon79, Jon81]. A graph-based formalism was chosen for two reasons:

- Graphs provide a reasonably compact formalism for describing heaps. Other techniques for reducing the number of nodes and edges in a memory graph are described in Chapter 6.
- Graphs are good tools for visualizing memory.

Later sections extend this notation with ideas from the literature on graph grammars and ideas from other papers on pointer-language analysis.

3. DEFINING DEPENDENCE

> *It is almost a defining characteristic of a 'mechanism' that, when it has produced a result, it is possible to inquire by what series of more elementary operations it has produced that result .... Since the primary purpose of a program is to specify a mechanism, it should be possible to associate with each program of a language a set of possible traces of the execution of that program; this association provides a 'mechanistic' formal definition of the language.* —C.A.R. Hoare [Hoa78]

A ***dependence*** is a relation that characterizes a program's behavior. The dependence $p \longrightarrow q$, roughly speaking, asserts that an execution of a statement p could interact with a subsequent execution of a statement q. Various algorithms that operate on programs use a program's dependences to determine how its statements *might* interact. These algorithms typically make valid (if conservative) judgments about a program's behavior when given a *safe estimate* (*i.e.*, a proper superset) of its dependences. This use of safe estimates is a concession to the limitations of static analysis: it is not possible to determine the set of dependences that an arbitrary program *must* exhibit (*cf.* §3.4.11).

A ***data dependence*** is a dependence that characterizes how a program operates on objects in memory—*i.e.*, structures, references, and streams. A data dependence $p \longrightarrow_d q$, roughly speaking, asserts that two statements p and q operate on a common object in memory. Simple and efficient algorithms exist for determining whether a given pair of statements might operate on a common object, relative to example languages in which every stored object has a unique name [Aho86]. The task of estimating a program's data dependences becomes more challenging in languages where objects do not have unique names—*e.g.*, in languages that support aliasing and dynamic allocation.

Chapters 3 though 6 develop algorithms for estimating a program's data dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$. These algorithms, roughly speaking, pair each of a program's statements with a set of ***abstract stores***. An abstract store is a special type of approximate memory configuration that represents a potentially infinite subset of *Store*—$\mathbf{M}_{\mathcal{H}}$'s domain of stores. The set of abstract stores that these algorithms pair with a statement q are a (finite) estimate of the (possibly infinite) set of stores that reach q during a computation. These algorithms also annotate every structure and reference *obj* in an abstract store with a *label*: a value that characterizes how a computation might have operated on *obj*. For example, the algorithm for computing a program's *flow* (*i.e.*, write-before-read) dependences labels every *obj* with those program points that might have defined *obj*'s value. This allows the set of flow dependences incident on a statement q to be estimated from the labels on the structures and references read at q.

The current chapter, which lays the groundwork for Chapters 4 through 6, defines notions of dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$. Section 3.1 presents an informal survey of the notion of dependence. Section 3.2 defines the basic types of dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$. Section 3.3 refines the notion of a data dependence. The refinements discussed in this section give a more precise picture of program evaluation in the presence of loops and procedures. Section 3.4 presents additional background on the notion of dependence.

Many of the concepts presented in Chapter 3 were originally developed by previous authors. Concepts that are original to this thesis include an alternative definition of def-order dependence (§3.2) and a generalized notion of loop-carried dependence for programs with procedures (§ 3.3).

3.1. An Informal Introduction to the Notion of Dependence

A dependence is a relation that characterizes how a program's statements interact over the course of its execution. Dependences are often used to determine whether the constraints on program execution that are inherent in a language's definition can be relaxed without altering a program's meaning. Dependences, for example, are frequently used to determine whether evaluating certain statements in parallel would change a program's meaning. Dependences are also used to identify and isolate a program's *slices*; *i.e.*, its logically related collections of statements [Ott84]. Other uses of dependences are given in the introduction, and mentioned throughout this chapter.

The paragraphs that follow present an informal taxonomy of dependence. The distinctions described below are important, since different kinds of dependences are used to reason about different aspects of program behavior. These distinctions are also fairly standard, up to subtle differences in assumptions about how dependences arise (*cf.* §3.4).

A program exhibits a ***control dependence*** $p \longrightarrow_c q$ when a point p determines whether a second point q executes. A program's control dependences are a reflection of a language's control structures. A program in an $\mathcal{H}$-like language, for example, may exhibit $p \longrightarrow_c q$ for any of the following reasons:

- p is a predicate that controls whether q evaluates. For example, q is control-dependent on p in the expression "[p] **if** *pred* **then** [q] $a := 1$ **fi**".
- p is the entry point of a procedure A, and q is a statement in A that is not enclosed by any loops or conditionals.
- p is a call to a procedure A, and q is A's entry point.

Dependences from call sites to procedure entry points are called ***interprocedural control dependences*** [Hor90a]. Other dependences are ***intraprocedural control dependences***.

A data dependence $p \longrightarrow_d q$ arises when points p and q manipulate common values. The dependence $p \longrightarrow_d q$ is typically classified according to how p and q interact. Most authors recognize four types of data dependences:

- A ***flow dependence*** $p \longrightarrow_f q$ arises when p writes to a location that q then reads.
- An ***anti-dependence*** $p \longrightarrow_a q$ arises when p reads from a location that q then overwrites.
- An ***output dependence*** $p \longrightarrow_o q$ arises when p writes to a location that q then overwrites.
- An ***input dependence*** $p \longrightarrow_i q$ arises when p reads from a location that q then reads.

A fifth type of data dependence, the ***def-order dependence*** is sometimes used instead of output dependence [Hor89]. These five dependences are illustrated in Figure 3.1. The distinctions between data dependences are important for analyzing program behavior: *e.g.*, for determining if an optimization is safe [Pad79, Kuh80, Kuc81, Wol82, All83, All87, Cal87, Fer87, Lar89, Bal89]. The following paragraphs give some reasons for these distinctions; more can be learned by consulting [Kuc81] or [Cal87].

Two statements p and q are said to exhibit a *read-write* conflict when one of these statements reads from, and the other writes to, a common location l. In a sequentially executed program, a read-write conflict that corresponds to a dependence can be classified as a flow or an anti-dependence. The distinction between

Input (Rd.-Before-Rd.)	*Anti* (Rd.-Before-Wr.)	*Flow* (Wr.-Before-Rd.)	*Output* (Wr.-Before-Wr.)
[1] $a := x$	[1] $a := x$	[1] $x := a$	[1] $x := a$
[2] $y := x$	[2] $x := y$	[2] $y := x$	[2] $x := y$
([1] $\longrightarrow_i$ [2])	([1] $\longrightarrow_a$ [2])	([1] $\longrightarrow_f$ [2])	([1] $\longrightarrow_o$ [2])

Def-Order

[1] $x := a$
[2] **if** *pred* **then**
[3] $x := b$
[4] $x := y$
fi
[5] $z := x$
([1] $\longrightarrow_{do([5])}$ [4])

A ***def-order dependence*** is a transitive output dependence (*i.e.*, a dependence of the form [p] $\longrightarrow_o \cdots \longrightarrow_o$ [q]) that is witnessed by a third program point.

The fifth example program exhibits [1] $\longrightarrow_o$ [3] $\longrightarrow_o$ [4].

The definitions at [1] and [4] are witnessed by the read at [5].

Figure 3.1. The five types of data dependence. All example dependences are through the variable x.

Program A_1	Program A_2	Program A_2 after parallelization
[1] $x := 2$;	[1] $tmp := 2$	**parbegin**
[2] $z := x$;	[2] $z := tmp$	**begin** [1] $tmp := 2$; [2] $z := tmp$ **end** ;
[3] $x := 1$;	[3] $x := 1$;	**begin** [3] $x := 1$; [4] $y := x$ **end**
[4] $y := x$;	[4] $y := x$;	**parend**

Figure 3.2. Using variable renaming to break an anti-dependence [2] $\longrightarrow_a$ [3] in Program A_1. This renaming allows statements in the resulting program, A_2, to be safely executed in parallel.

flow and anti-dependence is important because anti-dependences need not appear in certain representations of program behavior (*cf.* Chapter 7). This distinction is also important because anti-dependences can be eliminated by renaming variables. This idea is illustrated in Figure 3.2. Program A_1 in Figure 3.2 is not parallelizable, because [2] $\longrightarrow_a$ [3] prevents the simultaneous execution of ([1],[2]) and ([3],[4]). Dependence [2] $\longrightarrow_a$ [3], however, arises from the reuse of the variable x. Renaming one use of x to *tmp* eliminates these dependences—and yields the equivalent, parallelizable program A_2.

If a program redefines its variables, then some notion of output dependence is needed to characterize its execution. Output dependences *per se*, however, can sometimes represent needless constraints on a program's sequential evaluation. Def-order dependences were introduced by Horwitz, Prins, and Reps to obtain an alternative characterization of program behavior [Hor89]. A def-order dependence $p \longrightarrow_{do(r)} q$ is a transitive output dependence $p \longrightarrow_o^* q$ that satisfies the following conditions:

1. A flow dependence $p \longrightarrow_f r$ is transmitted through a location l.
2. A second flow dependence $q \longrightarrow_f r$ is transmitted through l.
3. p occurs to the left of q in a program's *abstract syntax tree* (*cf.* [Aho86]).

Intuitively, def-order dependences are used instead of output dependences because they constrain the sequence of values read from (rather than the sequence of values written to) a variable x (*cf.* Figure 3.3).

Input dependences are useful for program optimization when different sequences of memory accesses incur different costs. Kuck *et. al.*, for example, use input dependences to group statements that read the same array [Kuc81]. This optimization improves program performance by reducing how often large arrays are loaded into virtual memory.

This taxonomy of data dependence has emphasized dependences that result from operations on *stores*. Data dependences can also result from operations on *streams*. Consider, for example, the program "[1] **read**(w); [2] **read**(x);". This program exhibits a dependence [1] $\longrightarrow_d$ [2], since the read at [1] affects the value read at [2]. Some authors refer to such dependences as flow dependences [Hor89] or def-order dependences [Sel89]. Here, a dependence like [1] $\longrightarrow_d$ [2] is called a ***stream-mediated data dependence***; other terms such as "flow dependence" and "def-order dependence" are reserved for dependences that arise through stores.

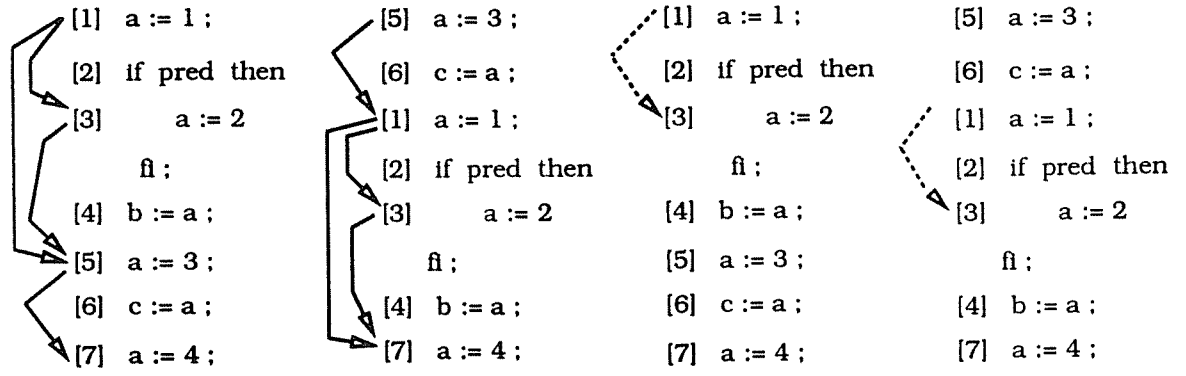


Figure 3.3. Programs that have inequivalent output dependences may have equivalent def-order dependences. Program P has three output dependences that program Q lacks: [1] $\longrightarrow_o$ [5], [3] $\longrightarrow_o$ [5], and [5] $\longrightarrow_o$ [7]. Similarly, Q has three output dependences that are missing from P. Programs P and Q, on the other hand, both have exactly one def-order dependence: [1] $\longrightarrow_{do(\{4\})}$ [3].

3.2. Definitions of Dependence for Language $\mathcal{H}$

3.2.1. Control dependence

Control dependence can be defined in terms of how a program *might* execute w.r.t. a given set of inputs. Such definitions of control dependence are useful, for example, in languages that support procedure-valued variables [Shi88]. The definition of control dependence used in this thesis, which is somewhat simpler, is based on the structure of a program's abstract syntax tree.

DEFINITION. A ***program point*** is a name that uniquely identifies a site in a program's abstract syntax tree. A program P has one program point for every *if* predicate, *while* predicate, assignment statement, and call statement that P contains. Program P also contains the following special program points:

- Points **initial**$_1$ and **initial**$_2$, which correspond to P's initial points of control. Point **initial**$_1$ initializes every object in a program's store. Point **initial**$_2$ invokes *main*( ).
- Point **final**, which corresponds to P's final point of control.
- For every call site "$[p]$ **call** $A\ (a_1, \cdots, a_n)$",
  - one point $[p.i_0]$ that saves the return address for the call to A;
  - n points $[p.i_1] \cdots [p.i_n]$ that compute the call's actual parameters.
- For every n-parameter procedure A,
  - a point $[A.enter]$ that represents A's entry point;
  - three points $[A.i_{-3}] \cdots [A.i_{-1}]$ that initialize A's local environment;
  - one point $[A.i_0]$ that saves a caller's return address;
  - n points $[A.i_1] \cdots [A.i_n]$ that initialize A's formal parameters; and
  - one point $[A.f]$ that represents the implicit return at the end of A. □

DEFINITION. Let q be a point in a program P. A *while* statement s ***encloses*** q if q is subordinate to s in P's abstract syntax tree. An *if* statement s ***encloses*** q if q is subordinate to s in P's abstract syntax tree. A call statement s ***encloses*** q if q is a special point that initializes one of s's actual parameters. □

DEFINITION. Let p and q be statements in a procedure P. Let $level(p)$ and $level(q)$ be the number of *call*, *while*, and *if* statements that enclose p and q, respectively. Statement q is (directly) ***control-dependent*** on p, written $p \rightarrow_c q$, iff either

1. p is the entry vertex, q is not the entry vertex, and $level(q) = 0$;
2. p is a call site, and p encloses q;
3. p is a *while* predicate, and $q = p$;
4. p is a *while* predicate, the *while* statement at p encloses q, and $level(q) = level(p) + 1$;
5. p is an *if* predicate, the **true** branch of the *if* statement at p encloses q, and $level(q) = level(p) + 1$; or
6. p is an *if* predicate, the **false** branch of the *if* statement at p encloses q, and $level(q) = level(p) + 1$.

Dependences that correspond to cases 1-5 are referred to as ***true-valued control dependences***. Dependences that correspond to case 6 are referred to as ***false-valued control dependences***. □

DEFINITION. Let p be a statement that calls procedure A. Let q be A's entry point. Then q is ***interprocedurally control-dependent*** on statement p. □

3.2.2. Data dependence

Data dependences arise through operations on streams and stores. The algorithms given in this thesis for computing a program's data dependences, however, ignore dependences that arise through streams. This decision follows from the observation that language $\mathcal{H}$ supports only one stream: the freelist. Chapter 7 argues that data dependences that arise from accesses of the freelist correspond to useless constraints on program behavior. Intuitively, a freelist-mediated data dependence, if honored, would restrict the order in which unreferenced structures were removed from a program's freelist. Language $\mathcal{H}$, however, is a ***referentially transparent*** language: none of its operators alter, or recognize specific, addresses in memory (*i.e.*, elements of domain *Loc*). Changes in how a program assigns addresses to newly allocated structures are therefore unobservable to the user.[2]

The definitions of store-mediated dependence given below are similar to—but not quite the same as—the ones presented in Section 3.1. The definitions given in Section 3.1 assume that statements perform only two kinds of operations on memory: *i.e.*, read and write location. These definitions work well for languages like FORTRAN, where a store's size is fixed throughout the execution of a given scope. They do not work as well for languages (like $\mathcal{H}$) whose operators can also alter the size of the store. For historical reasons, the terms *read* and *write* are used throughout this thesis to characterize how statements operate on memory. This circumlocution, however, requires a slight bending of the notion of a write.

DEFINITION (***write of a memory object***). A structure or reference is *written* when it is added to a store σ. □

DEFINITION (***write of a memory object at a state***). Let p be a program point, σ a store, and fl a freelist. A structure (reference) *obj* is *written at state* (p, σ, fl) iff the evaluation of p w.r.t. σ and fl writes *obj*. □

DEFINITION (***read of a memory object***). A structure or reference is *read* when it is accessed by the evaluation of an identifier expression. □

DEFINITION (***read of a memory object at a state***). Let p be a program point, σ a store, and fl a freelist. A structure (reference) *obj* is *read at state* (p, σ, fl) iff the evaluation of p w.r.t. σ and fl reads *obj*. □

Structures are written by the evaluation of *initialize* and assignment statements. The *initialize* statement "re-creates" every structure (and reference) in a program's initial store. The evaluation of "$\cdots := val$" adds the atom *val* to a store σ. The evaluation of "$\cdots :=$ **new**(*env*)" adds a new environment to σ. The evaluation of "$\cdots :=$ **new**(*typ*)", where *typ* is a user-defined structure with n fields, adds $n + 1$ structures to σ: one structure of type *typ*, and n **nil**-valued atoms. Finally, the evaluation of "$\cdots :=$ *saveContext*(*programPt*)" adds a special structure to σ that records a procedure call's return address.

---

[2] Admittedly, this is an unusual use of the term *referentially transparent*. Solomon [*private communication*] has suggested that the notion described here is akin to what the database community refers to as a "value-based" semantics—as opposed to a semantics where structures have distinct identities.

References are written by the evaluation of the *initialize* and assignment statements. The evaluation of "$\cdots$:= **new**(*typ*)", where *typ* ($\neq$ *env*) is a structure with n fields, adds n references to a store σ. The evaluation of "[*p*] *idexp* := $\cdots$", where $idexp = sel_1 \cdots sel_n$, adds a reference to a store σ. More specifically, let *gEnv* be σ's global environment. Let $idexpr(p, \sigma, gEnv, sel_1 \cdots sel_{n-1})$ denote the location l (*cf.* Appendix 1), and s the structure at location l. Then the evaluation of *idexp* := $\cdots$ adds a reference r at s of type sel_n.

DEFINITION. Let s be a structure and *sel* a selector. The ***reference of type sel at s*** is the reference at s that corresponds to *sel*. More precisely, let σ be a store such that $\sigma(loc) = s$; then the reference of type *sel* at s is the reference accessed by the evaluation of $selexp(\sigma, loc, sel)$. □

(*N.B.*: If there is already a reference r' at s of type sel_n, then the new reference r replaces r'.)

The set of structures and references that the evaluation of the identifier expression *idexp* reads varies according to *idexp*'s context. More specifically, let $idexp = sel_1 \cdots sel_n$ be an identifier expression at a point p. Let σ be a store and *gEnv* be σ's global environment. If *idexp* appears on the left-hand side of an assignment statement, then the evaluation of *idexp* at p reads those structures and references that are accessed by the evaluation of $idexpr(p, \sigma, gEnv, sel_1 \cdots sel_{n-1})$. Otherwise, the evaluation of *idexp* at p reads those structures and references that are accessed by the evaluation of $idexpr(p, \sigma, gEnv, sel_1 \cdots sel_n)$.

Technically, a structure s is also read when it is passed as an argument to *gettyp*, which returns s's type, or *getval*, which returns s's atomic value. The definition of $\mathbf{M}_{\mathcal{H}}$, however, ensures that any s passed to *gettyp* or *getval* must first be accessed by the evaluation of an identifier expression.

The informal definition of data dependence states that a dependences arises through successive operations on a common object in memory. This notion is formalized with a state transition relation, $\vdash$:

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

$$prog \vdash state_i \longrightarrow^0 state_j \Leftrightarrow state_j = state_i$$

$$prog \vdash state_i \longrightarrow^n state_j \Leftrightarrow \exists\, state' : prog \vdash state_i \longrightarrow^{n-1} state' \wedge state_j = evalPt(prog, state')$$

$$prog \vdash state_i \longrightarrow^* state_j \Leftrightarrow \exists\, n : prog \vdash state_i \longrightarrow^n state_j$$

$$prog \vdash state_i \longrightarrow^+ state_j \Leftrightarrow \exists\, n > 0 : prog \vdash state_i \longrightarrow^n state_j$$

$$prog \vdash state_n \longrightarrow \cdots \longrightarrow state_m \Leftrightarrow \forall i : n \le i \le m-1 : prog \vdash state_i \longrightarrow^1 state_{i+1}$$ □

The expression *evalPt*(*prog*, *state'*) constitutes a minor abuse of notation. The function *evalPt* (*cf.* Appendix 1) actually takes one formal parameter—a state—and three non-local parameters that describe *prog*'s control-flow graph, structure declarations, and local identifiers.

DEFINITION (***true for all states between ...***). A predicate $P : State \rightarrow Bool$ is *true for all states between* $state_n$ *and* $state_m$ iff $prog \vdash state_n \longrightarrow \cdots \longrightarrow state_m$ implies that $P(state_i) = true$ for all $i : n < i < m$. □

The definition of $\vdash$ may now be used to give formal definitions of flow, output, input, and anti-dependence:

DEFINITION (***flow dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Point q is (directly) *flow-dependent* on p w.r.t. *InSet*, written $p \longrightarrow_f q$ (w.r.t. *InSet*), iff there

exists a store $\sigma \in InSet$, a freelist fl, states (p, σ_p, fl_p) and (q, σ_q, fl_q), and an object obj such that

- $prog \vdash (\mathbf{initial}_1, \sigma, fl) \rightarrow^* (p, \sigma_p, fl_p) \rightarrow^+ (q, \sigma_p, fl_q)$;
- obj is written at (p, σ_p, fl_p);
- obj is not written at any states between (p, σ_p, fl_p) and (q, σ_q, fl_q); and
- obj is read at (q, σ_q, fl_q). □

DEFINITION (***input dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Point q is (directly) *input-dependent* on p w.r.t. *InSet*, written $p \rightarrow_i q$ (w.r.t. *InSet*), iff there exists a store $\sigma \in InSet$, a freelist fl, states (p, σ_p, fl_p) and (q, σ_q, fl_q), and an object obj such that

- $prog \vdash (\mathbf{initial}_1, \sigma, fl) \rightarrow^* (p, \sigma_p, fl_p) \rightarrow^+ (q, \sigma_p, fl_q)$;
- obj is read at (p, σ_p, fl_p);
- obj is not written at any states between (p, σ_p, fl_p) and (q, σ_q, fl_q); and
- obj is read at (q, σ_q, fl_q). □

DEFINITION (***output dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Point q is (directly) *output-dependent* on p w.r.t. *InSet*, written $p \rightarrow_o q$ (w.r.t. *InSet*), iff there exists a store $\sigma \in InSet$, a freelist fl, states (p, σ_p, fl_p) and (q, σ_q, fl_q), and a reference r such that

- $prog \vdash (\mathbf{initial}_1, \sigma, fl) \rightarrow^* (p, \sigma_p, fl_p) \rightarrow^+ (q, \sigma_p, fl_q)$;
- r is written at (p, σ_p, fl_p);
- r is not overwritten (*i.e.*, replaced) at any states between (p, σ_p, fl_p) and (q, σ_q, fl_q); and
- r is overwritten at (q, σ_q, fl_q). □

DEFINITION (***anti-dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Point q is (directly) *anti-dependent* on p w.r.t. *InSet*, written $p \rightarrow_a q$ (w.r.t. *InSet*), iff there exists a store $\sigma \in InSet$, a freelist fl, states (p, σ_p, fl_p) and (q, σ_q, fl_q), and a reference r such that

- $prog \vdash (\mathbf{initial}_1, \sigma, fl) \rightarrow^* (p, \sigma_p, fl_p) \rightarrow^+ (q, \sigma_p, fl_q)$;
- r is read at (p, σ_p, fl_p);
- r is not overwritten at any states between (p, σ_p, fl_p) and (q, σ_q, fl_q); and
- r is overwritten at (q, σ_q, fl_q). □

Output and anti-dependences can only arise through operations on references. The definition of $\mathbf{M}_{\mathcal{H}}$ does not allow the attributes of a structure s (*i.e.*, its type and atomic value) to be modified after s has been allocated. This observation is true, in part, because $\mathbf{M}_{\mathcal{H}}$ never returns structures that have been allocated to the freelist.

Section 3.1.2 states that a computation exhibits $p \rightarrow_{do(r)} q$ when it exhibits two flow dependences, $p \rightarrow_f r$ and $q \rightarrow_f r$, that arise through a common location l. This definition, when rephrased in terms of structures and references, states that a computation exhibits $p \rightarrow_{do(r)} q$ when it exhibits two flow dependences, $p \rightarrow_f r$ and $q \rightarrow_f r$, that arise through a common field in a common structure.

DEFINITION (***def-order dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Assume that p occurs to the left of q in *prog*'s abstract syntax tree. Point q is (directly) *def-order-dependent* on p w.r.t. *InSet* and r, written $p \rightarrow_{do(r)} q$ (w.r.t. *InSet*), iff there exists a store $\sigma \in InSet$, an initial freelist fl, four states (p, σ_p, fl_p), (q, σ_q, fl_q), (r, σ_r, fl_r), and (r, σ'_r, fl'_r), a pair of

structures s and s', and a selector *sel* such that

- $p \longrightarrow_f r$ w.r.t. $\{\sigma\}$ through a reference *ref* of type *sel* at structure s; *i.e.*,
  - *prog* $\vdash$ $(\textbf{initial}_1, \sigma, \mathit{fl}) \longrightarrow^* (p, \sigma_p, \mathit{fl}_p) \longrightarrow^+ (r, \sigma_r, \mathit{fl}_r)$;
  - *ref* is written at $(p, \sigma_p, \mathit{fl}_p)$;
  - *ref* is not overwritten at any states between $(p, \sigma_p, \mathit{fl}_p)$ and $(r, \sigma_r, \mathit{fl}_r)$; and
  - *ref* is read at $(r, \sigma_r, \mathit{fl}_r)$;
- $q \longrightarrow_f r$ w.r.t. $\{\sigma\}$ through a reference *ref'* of type *sel* at structure s'; *i.e.*,
  - *prog* $\vdash$ $(\textbf{initial}_1, \sigma, \mathit{fl}) \longrightarrow^* (q, \sigma_q, \mathit{fl}_q) \longrightarrow^+ (r, \sigma'_r, \mathit{fl}'_r)$;
  - *ref'* is written at $(q, \sigma_q, \mathit{fl}_q)$;
  - *ref'* is not overwritten at any states between $(q, \sigma_q, \mathit{fl}_q)$ and $(r, \sigma'_r, \mathit{fl}'_r)$; and
  - *ref'* is read at $(r, \sigma'_r, \mathit{fl}'_r)$; and
- s and s' are the same structure. □

The definition given above, however, is *not* a good starting point for determining a pointer program's def-order dependences. According to this definition, a computation c's def-order dependences can be determined if one knows whether s and s'—an *arbitrary* pair of structures that exist at *unrelated* moments $(r, \sigma_r, \mathit{fl}_r)$ and $(r, \sigma'_r, \mathit{fl}'_r)$ in c—are the same structure. To make such precise comparisons between arbitrary structures in σ_r and σ'_r possible, every structure that is allocated during the course of c must have a tag that uniquely distinguishes it from all other structures allocated during computation c. These tags could be implemented, for example, by pairing every structure s with an additional integer that identifies the moment in c at which s was allocated: *e.g.*, by pairing the kth structure allocated during the evaluation of c with the integer k. This assumption that every object initially has a unique identity, however, creates two problems for algorithms that analyze a pointer program's behavior.

- Tags are a potential source of imprecision. A language like $\mathcal{H}$ does not limit the number of structures that an arbitrary computation can allocate. There is therefore no *a priori* limit on the number of tags that a computation that paired structures with tags could require—even if the number of accessible structures in a computation's store is bounded. (This is true, for example, of a program such as "**while** *pred* **do** *a* := **new**(*conscell*) **od**".) To ensure that an analysis of an arbitrary computation terminates (*cf.* Chapter 5), the number of tags that a computation uses must somehow be limited: *e.g.*, by pairing the mth structure allocated by a computation with the value m *modulo* k for some predetermined constant k. This, however, can create spurious def-order dependences by (*e.g.*) causing the $m+k$ *modulo* kth structure allocated by a computation to be mistaken for the m *modulo* kth.
- Tags interfere with the elimination of redundant stores. Recall that language $\mathcal{H}$ is referentially transparent: two stores that are isomorphic up to how structures are paired with locations are indistinguishable from the standpoint of the language's operators—and are therefore interchangeable from the standpoint of dependence computation. Introducing tags into an analysis would make it more difficult to find opportunities for eliminating redundant stores. Two stores s and s' whose structures are allocated at different moments during a computation would never be indistinguishable for the purpose of dependence computation—unless, that is, the tags of s and s' are equivalent, relative to the chosen strategy for estimating tags.

The following, equivalent definition of def-order dependence eliminates these difficulties by rephrasing the notion of a def-order dependence in terms of a *single* sequence of states.

(RE)DEFINITION (***def-order dependence***). Let *prog* be a program with program points p and q, and *InSet* a set of stores. Assume that p occurs to the left of q in *prog*'s abstract syntax tree. Point q is (directly) *def-order-dependent* on p w.r.t. *InSet* and r, written $p \longrightarrow_{do(r)} q$ (w.r.t. *InSet*), iff there exists a store $\sigma \in InSet$, an initial freelist fl, states (x, σ_x, fl_x), $(r, \sigma_{r1}, fl_{r2})$, (y, σ_y, fl_w), and $(r, \sigma_{r2}, fl_{r2})$, a structure s, and a selector *sel* such that:

- $\{x, y\} = \{p, q\}$;
- $prog \vdash (\mathbf{initial}_1, \sigma, fl) \longrightarrow^* (x, \sigma_x, fl_x) \longrightarrow^+ (r, \sigma_{r1}, fl_{r1}) \longrightarrow^+ (y, \sigma_y, fl_y) \longrightarrow^+ (r, \sigma_{r1}, fl_{r2})$;
- a reference *ref* of type *sel* at structure s is written at (x, σ_x, fl_x);
- *ref* is not overwritten at any states between (x, σ_x, fl_x) and $(r, \sigma_{r1}, fl_{r1})$;
- *ref* is read at $(r, \sigma_{r1}, fl_{r1})$;
- a reference *ref'* of type *sel* at structure s is written at (y, σ_y, fl_y);
- *ref'* is not overwritten at any states between (y, σ_y, fl_y) and $(r, \sigma_{r2}, fl_{r2})$; and
- *ref'* is read at $(r, \sigma_{r2}, fl_{r2})$. □

According to this second definition, a program's def-order dependences can be computed by monitoring how a computation c performs sequences of operations on a store's component structures. This second definition is used in Chapter 4 to develop a strategy for determining def-order dependences that does not use tags to determine whether two structures are, in fact, the same structure.

3.3. Refining the Notion of Data Dependence

The dependence $p \longrightarrow_d q$ asserts that *any* execution of q might depend on *any* execution of p. Such an assertion is often an excessively weak estimate of program behavior. This is the case, for example, when p and q are statements in loops that access different elements of a common structure: *e.g.*, an array or list. Improved estimates of program behavior are often obtained by *refining* the notion of dependence—*i.e.*, by qualifying $p \longrightarrow_d q$ with assertions about *which* of p's and q's evaluations interact.

One such refinement, the *distance* of a data dependence, is discussed in Section 3.4.10. A second refinement of the notion of data dependence is the notion of a ***loop-carried dependence***. This concept was introduced by Allen [All83], who uses it to determine when nested loops can safely be interchanged. Roughly speaking, the assertion that a loop L *carries* $p \longrightarrow_d q$ corresponds to the assertion that $p \longrightarrow_d q$ arises from operations on L's induction variables. Figure 3.4 explains this notion by relating it to the data dependences exhibited by an unfolded loop. Assume that the example program in Figure 3.4 exhibits $p \longrightarrow_d q$, and that L runs for n iterations. Unfolding L n times yields an equivalent program that has one less loop and n copies of p and q. Let $[i, p]$ denote the copy of p produced by the ith unfolding of L. Assertions about whether L carries $p \longrightarrow_d q$ can be used to determine whether the unfolded program exhibits dependences of the form $[i, p] \longrightarrow_d [j, q]$:

- If $i \neq j$ and L does not carry $p \longrightarrow_d q$, then the unfolded program exhibits *no* dependences of the form $[i, p] \longrightarrow_d [j, q]$.
- If $i < j$ and L carries $p \longrightarrow_d q$, then $[i, p] \longrightarrow_d [j, q]$ is presumed to hold in the absence of special information about $[i, p]$ and $[j, q]$.

- The assertion that L carries $p \longrightarrow_d q$ says nothing about whether $[i, p] \longrightarrow_d [i, q]$.

A dependence $p \longrightarrow_d q$ is also said to be ***loop-independent*** if an $[i, q]$ depends on an $[i, p]$ after every loop in a set of loops has been *completely* unfolded.

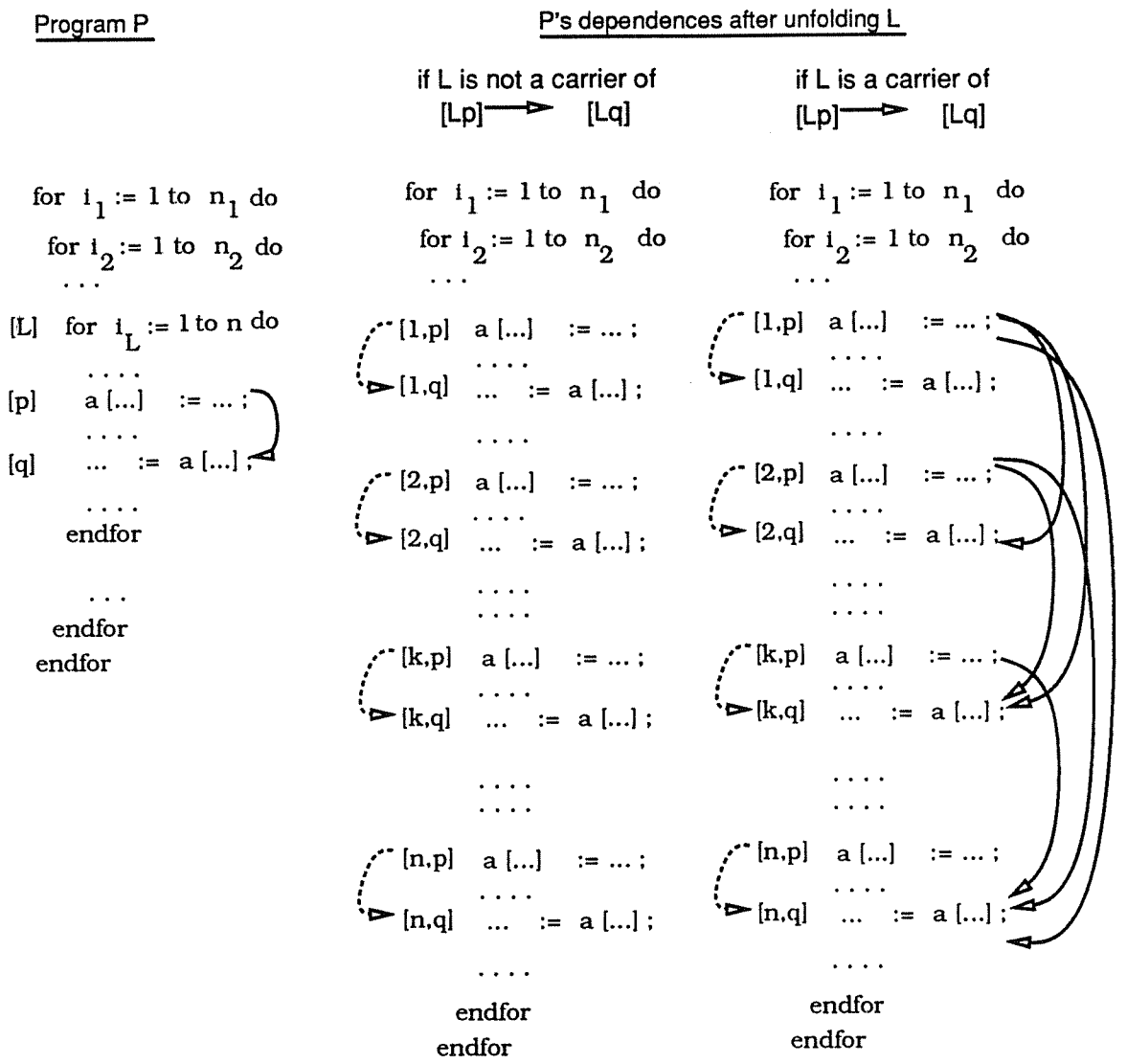


Figure 3.4. Loop-carried dependence. Solid arrows in unrolled programs denote dependences implied by assertion that L carries $p \longrightarrow_f q$. Broken lines denote possible dependences whose existence is independent of the assertion that L carries $p \longrightarrow_f q$.

```
   procedure main ()
[1]      call getX(1);
[2]      call putY(1);
[3]      call getX(2);
[4]      call putY(2);
...
[2n-1]   call getX(n);
[2n]     call putY(n);
   end;

   procedure getX (index)
[p]      temp[index] := x[index]
   end;

   procedure putY (index)
[q]      y[index] := temp[index]
   end;
```

There are n dependences from occurrences of p to occurrences of q:

$[\textbf{initial}_2\ 1, p] \rightarrow_f [\textbf{initial}_2\ 2, q]$

$[\textbf{initial}_2\ 3, p] \rightarrow_f [\textbf{initial}_2\ 4, q]$

...

$[\textbf{initial}_2\ 2n-1, p] \rightarrow_f [\textbf{initial}_2\ 2n, q]$

Figure 3.5. Using the Sharir-Pnueli call string notation to name a program's dependences.

Previous treatments of carriers have focused on languages with loops. This section develops a notion of a carrier for languages like $\mathcal{H}$ that support loops *and* procedures. This new notion of a carrier is defined as an approximation to the set of dependences that arise between *specific occurrences* (*i.e.*, distinct executions) of two program points. Assume, in other words, that there is some way of assigning a unique name $[n, x]$ to every occurrence of a program point $[x]$ in a computation c. Then this computation may be said to exhibit $[i, p] \rightarrow_d [j, q]$ iff

- $[i, p]$ and $[j, q]$ both occur in c;
- $[i, p]$ occurs before $[j, q]$ in c; and
- that part of c that runs from $[i, p]$ to $[j, q]$ exhibits $p \rightarrow_d q$.

The notion of a carrier will be defined in terms of the specific instances of p and q that exhibit $p \rightarrow_d q$.

Various schemes can be used to name a statement's occurrences. Because the notion of a carrier is closely connected with the notion of unfolding, it is important to use names that identify the circumstances under which these occurrences execute. Algorithms that analyze sets of nested loops, for example, typically use *iteration counts* to name the occurrences of a given statement. Under this naming scheme, each occurrence of statement [2] in the program "[1] **for** $i := 1$ **to** 64 **do** [2] $a[i+8] := a[i] * a[i]$ **od**" is numbered with the value of i at the time of that occurrence's evaluation. The program is then said to exhibit 56 flow dependences between the occurrences of [2]—namely, [1,2] $\rightarrow_f$ [9,2], [2,2] $\rightarrow_f$ [10,2], $\cdots$, [57,2] $\rightarrow_f$ [64,2]. (The observation that $j - i = 8$ for all $[i, 2] \rightarrow_f [j, 2]$ also allows this loop to be split into eight parallelizable loops.)

A more general naming scheme is needed to distinguish between different occurrences of statements in languages with loops and procedure calls. A satisfactory naming scheme for such languages can be developed from another naming device—the ***call string*** [Sha81]. Intuitively, a call string is an abstraction of a program's stack that names a program's active call sites.[3] Figure 3.5 shows how call strings can be

[3] A similar naming device appears in Harrison's thesis on parallelizing Scheme (*cf.* §2.6, [Har89]). Harrison's *procedure strings* are a variant of call strings that name call sites *and* procedures. The need to include procedure names in procedure strings stems from the presence of procedure-valued variables in Scheme.

used to name the dependences that arise between individual occurrences of a program's statements. The program shown, program P, is a program in an extended version of $\mathcal{H}$ that, for the sake of example, also supports arrays. The string "$\mathbf{initial}_2\ 2i-1\ p$" names that occurrence of p that sets $temp[i]$ to i. (*N.B.*: $\mathbf{initial}_2$ is the first element of every call string because *main*( ) is first called at $\mathbf{initial}_2$.) The string "$\mathbf{initial}_2\ 2i\ q$" names that occurrence of q that sets $x[i]$ to i.

Sharir and Pnueli use call strings to determine how specific procedure calls affect a program's store. The naming mechanism used in this thesis, the ***occurrence string***, is a call string that also records the evaluation of a program's loops. Occurrence strings are illustrated in Figure 3.6. The program in Figure 3.6, which is similar to the one in Figure 3.5, uses a loop to transfer data between arrays x and y. This program's occurrence strings record snapshots of that program's stack. These occurrence strings *also* record how many times the loop at statement [2] has evaluated—if this loop has not yet finished running. This information about loop evaluation is needed to distinguish among the different invocations of the example program's auxiliary procedures. In particular, "$\mathbf{initial}_2\ 2^i\ 3\ p$" names that occurrence of p that sets $temp[i]$ to i. Similarly, "$\mathbf{initial}_2\ 2^i\ 4\ q$" names that occurrence of q that sets $y[i]$ to $temp[i]$. (*N.B.*: m^k denotes a k-long sequence of m's.)

Occurrence strings can be used to develop occurrence-specific definitions of dependence for language $\mathcal{H}$. Let $t = (p_1, \sigma_1, fl_1) \cdots (p_k, \sigma_k, fl_k) \cdots$ be the ***trace*** of an example computation:

> DEFINITION (***trace of a computation***). Let *prog* be a program and σ a store. The *trace* of *prog* on σ is the sequence $(p_1, \sigma_1, fl_1) \cdots (p_n, \sigma_n, fl_n) \cdots$, where $p_1 = \mathbf{initial}_1$, $\sigma_1 = \sigma$, fl_1 is the freelist that $\mathbf{M}_{\mathcal{H}}$ pairs with σ, and, for all i, $prog \vdash (p_i, \sigma_i, fl_i) \longrightarrow (p_{i+1}, \sigma_{i+1}, fl_{i+1})$. □

The following algorithm, adapted from similar algorithms in Sharir and Pnueli (§ 7.3, *ibid.*) and Harrison [Har89], computes *occurrence* (t, n), the occurrence string for the program point at the nth state in t:

- *occurrence* $(t, 1)$ is ε, the empty occurrence string.
- If $n > 1$, let $o = o_1 \cdots o_k$ be *occurrence* $(t, n-1)$. *occurrence* (t, n) may now be computed from o, p_{n-1}, and p_n, as follows:

```
   procedure main ( )
[1]   i := 1 ;
[2]   while i ≤ n do
[3]      call getX(i);
[4]      call putY(i);
[5]      i := i + 1 ;
      end;
   end;

      procedure getX (index)
[p]   temp[index] := x[index]
      end;

      procedure putY (index)
[q]   y[index] := temp[index]
      end;
```

There are n dependences from occurrences of p to occurrences of q:

$[\mathbf{initial}_2\ 2\ 3, p] \longrightarrow_f [\mathbf{initial}_2\ 2\ 4, q]$

$[\mathbf{initial}_2\ 2\ 2\ 3, p] \longrightarrow_f [\mathbf{initial}_2\ 2\ 2\ 4, q]$

...

$[\mathbf{initial}_2\ 2^n\ 3, p] \longrightarrow_f [\mathbf{initial}_2\ 2^n\ 4, q]$

Figure 3.6. Using occurrence strings to name a program's dependences.

* If (p_{n-1}, p_n) is an entry arc into a procedure A, then *occurrence* $(t) = o_1 \cdots o_k A$.
* If (p_{n-1}, p_n) is an entry arc into the *body* of a loop L, then *occurrence* $(t) = o_1 \cdots o_k L$.
* If (p_{n-1}, p_n) is an exit from a procedure, then *occurrence* $(t) = o_1 \cdots o_{k-1}$.
* If (p_{n-1}, p_n) is an exit from a loop L—that is, an arc from the loop's predicate to the first statement following L—then o must have been of the form $o_1 \cdots o_j L^{k-j}$, where $o_j \neq L$ and j is the number of consecutive iterations of L that have completed. *occurrence* (t) is then $o_1 \cdots o_j$. □

Intuitively, this algorithm computes occurrence strings by reducing a prefix of a program's trace to a string of active procedure calls and loop invocations. The relationship between this algorithm and the Sharir-Pnueli and Harrison algorithms becomes evident if one thinks of each invocation of a loop as a call on a tail-recursive procedure.

The notion of a *carrier* of a dependence $p \longrightarrow_d q$ may now be defined as an abstraction of the set of all dependences between specific occurrences of p and q that a program exhibits.

DEFINITION (***carriers of an occurrence-specific dependence***). Let $d = [i, p] \longrightarrow_d [j, q]$, where i and j are occurrence strings, be a dependence between two specific occurrences of a pair of program points, p and q. Let i and j be strings of the form $p_1 \cdots p_x i'_1 \cdots i'_l$ and $p_1 \cdots p_x j'_1 \cdots j'_m$, where $p_1 \cdots p_x$ is the longest common prefix of i and j. The ***carrier*** of d, written *carrier* (d), is $(\varepsilon, \varepsilon)$, if $l = m = 0$; (ε, j'_1), if $l = 0$ and $m > 0$; (i'_1, ε), if $l > 0$ and $m = 0$; and (i'_1, j'_1), if $l > 0$ and $m > 0$. □

DEFINITION (***carrier-independent dependence***). An occurrence-specific data dependence is *carrier-independent* if its carrier is $(\varepsilon, \varepsilon)$. □

DEFINITION (***loop-carried dependence***). Let l be the entry point of a loop. An occurrence-specific data dependence is *carried by* l if its carrier is either (ε, l) or (l, m), where m is transitively control-dependent on l, and l is *not* transitively control-dependent on m. An occurrence-specific data dependence d is *carried by a loop* if there exists an l that carries d. □

DEFINITION (***call-site-carried dependence***). An occurrence-specific data dependence is *carried by a call site* if its carrier is (x, y), where either x or y is a call site. □

Figures 3.5 and 3.6 depict call-site carried dependences. In Figure 3.5, the dependence $[\mathbf{initial}_2\, 2i-1, p] \longrightarrow_f [\mathbf{initial}_2\, 2i, q]$ is carried by ([2*i*−1],[2*i*]). In Figure 3.6, every dependence $[\mathbf{initial}_2\, 2^i\, 3, p] \longrightarrow_f [\mathbf{initial}_2\, 2^i\, 4, q]$ is carried by ([3],[4]). Examples of carrier-independent dependences and loop-carried dependences are given in Figure 3.7.

DEFINITION (***carriers of a dependence***). The *carriers* of a dependence $p \longrightarrow_d q$ are the set of all *carrier* $([i, p], [j, q])$ such that $[i, p] \longrightarrow_d [j, q]$. □

Since programs are finite syntactic objects, *carriers* $(p \longrightarrow_d q)$ is always a finite set.

A use of the notion of a call-site-carried dependence is illustrated in Figure 3.8. The first program in Figure 3.8, program P, is the program depicted in Figure 3.5. The second program in Figure 3.8, program Q, is a permutation of P. A theorem proved in Section 7.3 states that P and Q, roughly speaking, represent equivalent programs if they have equivalent control, flow, and def-order dependences. A naive characterization of these program's dependences—*i.e.*, one that fails to use the notion of the carriers of a dependence—suggests that the set of flow dependences in P and Q are not the same. For example, the

```
   struct intptr is <intp>;

   procedure main()
      value := new(intptr);
[p]   read(value.intp);
[s1]  call conditionalPrint(0, value);
[s2]  call conditionalPrint(1, value);
[I]   for i := 1 to 2 do
[m]    for j := 1 to 2 do
[m1]    read(y[i+1, 1]);
[m2]    k := y[i, j];
[m3]    x[i, j] := k;
       endfor
      endfor
   end;

   procedure conditionalPrint(printswitch, ptri)
     if printswitch ?= 1 then
[q]    print(ptri.intp)
     fi
   end;
```

$[p] \longrightarrow_f [q]$ is carried by $[s_2]$ (*i.e.*, by $(\varepsilon, [s_2])$).

$[m_1] \longrightarrow_f [m_2]$ is carried by $[I]$ (*i.e.*, by $([m],[I])$).

$[m_2] \longrightarrow_f [m_3]$ is carrier-independent.

Figure 3.7. More examples that illustrate of the notion of a carrier. Here, the language has been extended, for the sake of example, to include arrays, **for** statements, **read** statements, and **print** statements.

Program *P*

```
   procedure main()
[1]    call getX(1);
[2]    call putY(1);
[3]    call getX(2);
[4]    call putY(2);
...
[2n-1] call getX(n);
[2n]   call putY(n);
   end;

   procedure getX(i)
[p]    temp[i] := x[i]
   end;

   procedure putY(i)
[q]    y[i] := temp[i]
   end;
```

Program *Q*

```
   procedure main()
[2n-1] call getX(n);
[2n]   call putY(n);
...
[3]    call getX(2);
[4]    call putY(2);
[1]    call getX(1);
[2]    call putY(1);
   end;

   procedure getX(i)
[p]    temp[i] := x[i]
   end;

   procedure putY(i)
[q]    y[i] := temp[i]
   end;
```

In both programs, the dependence $p \longrightarrow_f q$ is carried by the call site pairs ([1],[2]), ([3],[4]), $\cdots$ ([2*n*-*I*], [2*n*]).

Figure 3.8. A use of the notion of carrier to show that two programs have equivalent (flow) dependences.

occurrence of q evaluated during the final call to *putY* in both programs appears to depend on all preceding evaluations of p. A more careful characterization of $p \longrightarrow_f q$—one which notes that the invocation of q at point $[2i]$ depends only on the invocation of p at $[2i-1]$—is needed to establish that P and Q have equivalent flow dependences.

3.4. Additional Background on the Notion of Dependence

Section 3.4 presents additional material on the notion of dependence. This material was not presented in Sections 3.1 through 3.3 because none of these topics are explored in later sections of the thesis.

Section 3.4.1 gives an informal history of the notion of dependence. Sections 3.4.2 through 3.4.11 discuss related notions of program behavior. This includes other notions of dependence, other kinds of dependences, and other dependence-like notions of statement interaction. Section 3.4.12 concludes with observations on the limitations of the notion of dependence.

3.4.1. Historical background

The concept of dependence grew out of work in the 1960's on the parallelization of FORTRAN. The apparent predecessor of the notion of a data dependence is the notion of a conflict (*cf.* §3.4.3). This notion appears in a seminal paper by Bernstein, who proved that two statements p and q could be executed in parallel whenever p and q do not conflict [Ber66].

Notions similar to control and data dependence are proposed in a 1970 paper by Tjaden and Flynn [Tja70]. Tjaden and Flynn, who discuss algorithms for parallelizing assembly language programs, use three notions to characterize interactions between a program's statements. The first, a ***procedural dependency***, is analogous to a control dependence. The second, a ***data dependency***, corresponds to a flow dependence. The third, an ***operational dependency***, corresponds to a busy-wait. Tjaden and Flynn sketch an algorithm for determining when additional registers can be used to break read-write and write-write conflicts between pairs of statements. This algorithm is analogous to later algorithms that use variable renaming to break anti- and output dependences.

Most of the common terms for describing dependence were developed during the 1970's at the University of Illinois:

A 1972 paper by Kuck, Muraoka, and Chen draws an explicit distinction between *flow dependences* and *anti-dependences*, there called forward and backward dependences, respectively [Kuc72]. This paper also defines ***distance vectors***—n-tuples that give the distance of a dependence (in n-space) w.r.t. a collection of n nested loops. Kuhn [Kuh80] credits Muraoka [Mur71] with the development of the notion of a distance vector.

Towle appears to have introduced the notions of *output dependence* and *transitive dependence* (referred to by Towle as indirect data dependence) [Tow76]. The word *appears* is used, since Towle was not

---

```
[1] for m := 1 to 2 do
[2]   for n := 1 to 2 do
[3]     read(a[m+1, 1]);
[4]     print(a[m, n]);
    endfor*
```

This example program exhibits [3] $\longrightarrow_f$ [4]. According to Allen's definition, [3] $\longrightarrow_f$ [4] is carried by the loop at [1]. According to Horwitz *et. al*'s definition, [3] $\longrightarrow_f$ [4] is carried by the loops at [1] and [2].

Figure 3.9. Illustrating the difference between Horwitz *et. al.*'s and Allen's definitions of loop-carried dependence.

careful to distinguish original concepts from concepts that were borrowed from earlier authors.

The notion of a ***direction vector*** was first proposed by Wolfe [Wol78]. A direction vector, a vector that gives the signs of a distance vector's entries, plays an important role in loop interchange.

Several authors, including Wolfe, credit Kuck's text on compiler-writing for the notion of input dependence [Kuc78].

Allen's 1983 thesis [All83] introduced the notions of *loop-carried* and *loop-independent dependence*. Allen used these notions to determine the safety of loop interchange. More recently, a different definition of loop-carried and loop-independent dependence was given by Horwitz, Prins, and Reps [Hor89]. Horwitz *et. al.* state that a dependence $p \longrightarrow q$ is carried by a loop L when a path in the control-flow graph that gives rise to $p \longrightarrow q$ includes a backedge to L's entry point. This definition provides an adequate, but less precise, characterization of a program's evaluation. Horwitz *et. al.*'s definition implies that a loop L carries a dependence d whenever a loop L' that encloses L carries d—even when L, by Allen's definition, does not (*cf.* Figure 3.9).

The notion of a *def-order* dependence was introduced by Horwitz, Prins, and Reps in the context of program integration [Hor87].

3.4.2. Def-use chains, support sets, and dominance

A dependence is one of several relations that have been used to characterize program behavior. Notions that are equivalent to the notion of flow dependence include the standard dataflow notion of a ***def-use chain*** (*cf.* [Aho86]) and Neirynck's notion of an expression's ***support*** [Nei88].

The notion of control dependence is closely related to the notions of ***dominance*** and ***post-dominance***. A node n in a program's control-flow graph dominates a second node n' if all paths in the graph from the entry point to n' pass through n. Similarly, a node n is post-dominated by a second node n' if all paths from n to the program's exit point pass through n'. Ferrante, Ottenstein, and Warren give a definition of control dependence that also extends to languages with gotos [Fer87]. This definition states that statement q is control-dependent on predicate p iff

- there exists a path π in a program's control-flow graph from p to q such that every point along this path, p and q excepted, is post-dominated by q, and
- p is not post-dominated by q.

3.4.3. Conflicts

Another notion that is closely related to the notion of data dependence is the notion of a *conflict*. Two statements p and q conflict, written $p \leftrightarrow q$, if both access the same memory location l and either updates l. The difference between a conflict and a dependence (which is also discussed in Section 3.1) is illustrated by the following example program:

[1] $x := 10$; [2] $x := 20$; [3] $y := x$; [4] $x := 40$;

This program exhibits three read/write conflicts and three write/write conflicts: [1] $\leftrightarrow_{rw}$ [3], [2] $\leftrightarrow_{rw}$ [3], [3] $\leftrightarrow_{rw}$ [4], [1] $\leftrightarrow_{ww}$ [2], [1] $\leftrightarrow_{ww}$ [4], and [2] $\leftrightarrow_{rw}$ [4]. It also exhibits four data dependences: [1] $\longrightarrow_{o}$ [2], [2] $\longrightarrow_{f}$ [3], [3] $\longrightarrow_{a}$ [4], and [2] $\longrightarrow_{o}$ [4].

Good results have been obtained from using conflicts to guide program parallelization (*e.g.*, [Hen90]). Even so, the notion of conflict equivalence has shortcomings that make it unsuitable for reasoning about certain aspects of program behavior. Consider, for example, the information that conflicts give about the following programs:

Program A_1:			Program A_2:		
	[1]	$x := 1$;		[3]	$x := 2$;
	[2]	$y := x$;		[4]	$z := x$;
	[3]	$x := 2$;		[1]	$x := 1$;
	[4]	$z := x$;		[2]	$y := x$;
	[5]	$x := 3$;		[5]	$x := 3$;

Programs A_1 and A_2 compute the same final values for x, y, and z. Neverthless, A_1 and A_2 are not conflict-equivalent; the two programs perform different sequences of updates to x. This example, like Figure 3.3, suggests why conflict analysis is not a good starting point for analyzing a program's meaning.

Conflict analysis plays an important role in the detection of program anomalies that arise from concurrent program execution. One such anomaly, a ***race condition***, is illustrated by the example program "**parbegin** $x := f()$; $x := g()$ **parend** ; $y := x$;". If $f()$ and $g()$ return different values, then the final values of x and y depends on which assignment to x completes first.

Taylor and Osterweil were among the first to use dataflow analysis to detect potential race conditions (and other anomalies of concurrent programs) at compile-time [Tay80]. Balasundaram and Kennedy, who also use conflict analysis to detect race conditions, observe that conflict analysis is the proper starting point for detecting these anomalies [Bal89a]. More specifically, they note that the notion of dependence is not well-defined when there is no *a priori* order on statement evaluation. Recent work by Netzer and Miller uses a combination of static and run-time techniques to identify an execution's race conditions [Net91, Net91a].

3.4.4. Logic-based and denotational notions of dependence

This thesis defines an operational notion of dependence. The notion of data dependence also arises in logic programming, where clauses that use a variable are said to be dependent on other clauses that define that variable (*cf.* [Deb89]). A third, *denotational* approach to defining the notion of dependence is described in a report by Hudak and Young [Hud91]. Hudak and Young state that an expression $expr_1$ is dependent on an expression $expr_2$ w.r.t. an environment *env* if the evaluation of $expr_1$ relative to *env* is affected by $expr_2$'s meaning. To be precise, they define $expr_1$ to be dependent on $expr_2$ w.r.t. *env* if a "booby-trapping" of the language's semantic function F that causes F to fail uniformly at $expr_1$ w.r.t. *env* changes the meaning of $expr_2$. This particular definition of dependence was chosen, in part, because computations in functional languages have no internal state. To investigate whether a given expression contributes to a program's meaning, one must perturb that program's meaning function.

3.4.5. Semantic dependence

A notion of dependence that is similar to Hudak and Young's is defined by Podgurski and Clarke [Pod90]. This dependence, the ***semantic dependence***, corresponds to the assertion that changing an operator in statement p might affect the sequence of values produced at statement q. Podgurski and Clarke use this notion

to discuss the problem of determining how a typographical error at a statement p might effect the evaluation of a statement q. Other theorems are given that relate the notions of semantic dependences and control and data dependence.

3.4.6. Imperative dependence

Pingali *et. al.* use the notion of an ***imperative dependence*** to model the evaluation of standard, imperative languages [Pin91]. Intuitively, an imperative dependence asserts that a pair of statements like "*load x* ; *test x*" must evaluate in a specific order. This notion allows Pingali *et. al.* to define a type of dependence graph, the *dependence flow graph*, that can be used to execute FORTRAN programs (*cf.* Chapter 7).

3.4.7. Weak control dependence

Most authors ignore the effect of *execution anomalies* (*i.e.*, errors, points of nontermination) on a program's flow of control. Podgurski and Clarke, on the other hand, argue that anomalies create a second type of control dependence, which they call the ***weak control dependence*** [Pod90]. A weak control dependence arises when one statement suppresses a second statement's execution. For example, the program "[1] **while true do skip od** ; [2] **print**('done')" exhibits the weak control dependence [1] $\longrightarrow_{wc}$ [2], since statement [1] fails to terminate. Podgurski and Clarke argue that weak control dependences are important for proper debugging and testing. For example, an optimizer that ignored [1] $\longrightarrow_{wc}$ [2] might generate an executable image that printed "done"—thereby surprising a person who was debugging this program.

3.4.8. Approximate notions of data dependence

Dependence is sometimes defined w.r.t. simplified models of computation. One common definition of dependence, which resembles the definition of a def-use chain, states that $p \longrightarrow_d q$ through x in a program P if p and q manipulate x, and there is an x-definition-free path in P's (extended) control-flow graph (*cf.* [Aho86]). A slighly more precise formulation of this definition states that a program exhibits $p \longrightarrow_f q$ if (1) paths in its control-flow graph link p to q, and (2) p and q might access a common location [All83]. Both definitions are static approximations to the notion of dependence defined in Section 3.2.2.

3.4.9. Dependences and intended behavior

Yet another area of concern is whether the notion of a dependence should be based on a more resilient model of computation—one that captures a program's ***intended behavior***. This concern stems from the observation that an incomplete or erroneous program can contain well defined threads of computation. Consider, for example, the following program, P:

[1] $y := 1/0$; [2] $x := 1$; [3] **print**(x)

Under the standard, *control-driven* model of program execution, P exhibits *no* data dependences: it simply halts before completing statement [1]. This characterization of P's dependences, however, is useless to the programmer who wishes to view P as a program in the making. Such a programmer would probably determine P's dependences by ignoring the error at [1]. Under this alternative model of evaluation, P exhibits [2] $\longrightarrow_f$ [3]. Similar observations could be used to infer that P is equivalent to

[1] $z := 1/0$; [2] $x := 1$; [3] **print**(x);

but not to

[1] $z := 1/0$; [2] **print**(x); [3] $x := 1$;

Improvements to the control-driven model have been proposed that provide more aggressive characterizations of a program's threads of computation. Two alternative models of computation are discussed in a paper by Felleisen and Cartwright [Car89]. The one, ***lackadaisical evaluation***, allows one thread of computation to continue evaluating if another, unrelated thread of computation fails. The other, ***lazy evaluation***, allows a program to continue to run even if all of its threads fail. For other recent discussions of alternative models of computation, see [Bal90] and [Pin91].

3.4.10. Distance of a dependence

Section 3.3 describes one refinement of the notion of data dependence. A second important refinement is the notion of a dependence's ***distances***. A dependence $p \longrightarrow_d q$ exhibits a distance d w.r.t. a loop L iff the memory operations that create $p \longrightarrow_d q$ could span d iterations of L. Distance plays a crucial role in discovering which invocations of a loop may be run in parallel [Lam74, Wol82, Kuh80, All83, All87, Cal87, Lar89, Bal89]. Consider, for example, the problem of parallelizing the program "[1] **for** $i := 1$ **to** 64 **do** [2] $a[i+8] := a[i] * a[i]$ **od**". A compiler seeking to parallelize loop [1] needs to know that [2] $\longrightarrow_{lc([1])}$ [2]—*i.e.*, that values produced by earlier evaluations of [2] are used by later evaluations of [2]. However, the assertion [2] $\longrightarrow_{lc([1])}$ [2] is an excessively cautious characterization of how this program runs. Of more use is the observation that the element of a produced by the ith occurrence of [2] is used only by the $i+8$th occurrence of [2]. This second observation yields an equivalent program that runs in one-eighth of the time:

```
parbegin
  for i := 1 to 64 step 8 do a[i+8] := a[i] * a[i] od;
  ...
  for i := 8 to 64 step 8 do a[i+8] := a[i] * a[i] od
parend
```

Authors who have given techniques for computing a dependence's distance w.r.t. a language with dynamic allocation include Horwitz, Pfeiffer, and Reps [Hor89a]; Bodin [Bod90]; Gohkale and Smith [Goh90]; and Larus [Lar89]. The first three of these reports are concerned with intraprocedural dependence computation. They propose algorithms that a compute a dependence's distance w.r.t. a set of nested loops. Larus, who describes techniques for parallelizing a series of recursive calls to the same procedure, was apparently the first to use the notion of distance to characterize interprocedural dependence.

3.4.11. Declaration dependence

Languages like $\mathcal{H}$ that provide type and variable declarations exhibit a third type of dependence. A ***declaration dependence*** $p \longrightarrow_{decl} q$ arises when a point q uses information about a program's types or structures given in a declaration p (*cf.* Figure 3.10). Declaration dependences are regarded as distinct from data dependences, since the information being transferred is not a computable value.

Declaration dependences are rarely mentioned in the literature on dependences: concerns about a program's declarations simply do not arise in most treatments of program behavior. One exception is a paper by Hood, Kennedy, and Müller that deals with efficient module recompilation [Hoo86]. The notion

```
[d1]  struct conscell is <hd, tl>;
[d2]  struct intptr is <intp>;

      procedure main ()
[d3]   local cell;
[1]      cell := new (conscell);
[2]      value := new (intptr);
[3]      cell.hd := 1;
[4]      value.intp := 2;
      end
```

[d1] $\rightarrow_{decl}$ [1], since **new** uses *conscell*'s definition to create a structure.
[d1] $\rightarrow_{decl}$ [3], since statement [3] checks if *hd* can be applied to *cell*.
[d2] $\rightarrow_{decl}$ [2], since **new** uses *intptr*'s definition to create a structure.
[d2] $\rightarrow_{decl}$ [4], since statement [4] checks if *intp* can be applied to *value*.
[d3] $\rightarrow_{decl}$ [1] and [d3] $\rightarrow_{decl}$ [3],
since [d3] declares *cell* to be a local variable.

Figure 3.10. Declaration dependence.

of intermodular dependence developed in this paper is similar to the concept of declaration dependence sketched above.

Declaration dependences are ignored in this thesis. To simplify the presentation, it is simply assumed that *each* of a program's statements is declaration-dependent on *all* of its declarations. This is clearly a pessimistic characterization of a typical program's declaration dependences. It should be equally clear that better characterizations are easy to obtain when each of a program's structures has a unique set of field names. Consider, for example, the program in Figure 3.10. It is easy to see that statement [1], which manipulates an object of type *conscell*, is not dependent on declaration [*d2*], which defines an object of type *intptr*. The problem of computing an arbitrary program's declaration dependences, however, becomes harder when fields in different structures have the same names. Consider, for example, the following program:

```
[d1] struct stoplight is <color, timer> ;  [d2] struct hat is <color, kind, size> ;
...
[p]  thing.color := red ;
```

Without more information about the referents of *thing*, it is impossible to tell whether p is dependent on $d1$, or $d2$, or on both declarations.

3.4.12. Limitations of dependence

Section 3.1 gave reasons for using dependences to analyze programs. There are also important facts about program behavior that dependences do not provide. One such fact is the specific values that a pair of program points share. Dependences alone, for example, cannot be used to determine the equivalence of "**if** *pred* **then** $x := 1$ **else** $x := 0$ **fi**; $y := x$" and "$x := 0$; **if** *pred* **then** $x := 1$ **fi**; $y := x$". Algorithms, however, have been given that use dependences to make more complex judgments about a program's meaning. One such algorithm, given by Yang, *can* determine that these two programs are equivalent (*cf.* Chapter 4, [Yan90]).

A second limitation of dependence is that an arbitrary program's dynamic dependences are uncomputable. This assertion follows from the fact that it cannot be determined whether arbitrary statement must evaluate [Man74]. It is, however, possible to design algorithms that compute safe estimates (*i.e.*, proper

supersets) of a program's dependences. Various researchers have also explored the use of special tools and language constructs that allow users to state that a possible dependence will not, in fact, be exhibited by any possible execution of a given program (*e.g.*, [All86, Die87, Lar89]).

4. USING AN INSTRUMENTED SEMANTICS TO CHARACTERIZE DATA DEPENDENCE

> *SETL is a set-theoretically oriented language of very high level whose repertoire of semantic objects includes finite sets, ordered* n*-tuples, and sets of ordered* n*-tuples usable as mappings. This two-part paper studies some of the optimization problems associated with such a language. The first issue studied is that of* copy optimization, *i.e. the discovery of cases in which change to a compound object can be made 'differentially' without recopying the whole of the object. This optimization is brought to rest on an analysis of* value flow, *i.e., on an analysis which finds all the points* p *in a SETL program at which the object created or modified at another point* q *can reappear.* —J. Schwartz [Sch75]

In Chapter 3, the notion of a data dependence was defined in terms of the *sequences* of states that a computation might generate w.r.t. $\mathbf{M}_{\mathcal{H}}$. Chapter 4 develops alternative definitions of data dependence that simplify the task of dependence computation. These new definitions treat dependence as a function of the *set* of states that a computation might generate, relative to a collection of non-standard, *instrumented semantics* for $\mathcal{H}$. These semantics, roughly speaking, label every structure and reference *obj* with the names of those statements that have read and written *obj*. These labels allow the data dependences that are incident on a statement q to be determined from the labels on the objects that are read—and overwritten—at q.

Chapter 4 is divided into three sections. Section 4.1 sketches a set of labeling techniques for characterizing a program's data dependences. Section 4.2 argues that the definitions of flow dependence given in Section 4.1 are equivalent to the (sequence-based) definitions of dependence given in Chapter 3. The first part of Section 4.2 uses an instrumented semantics for $\mathcal{H}$, $\mathbf{MI}_{\mathcal{H}}$, to formalize the notion of flow dependence. The second part of Section 4.2 uses a lemma that relates $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ to establish that the two semantics' characterizations of flow dependence are equivalent. (Similar arguments show the equivalence of the remaining notions of data dependence.) Section 4.3 then concludes with a discussion of related work.

Instrumented semantics have played an important role in program analysis since the early 1970's. This chapter's specific contributions include the use of labels to characterize a pointer program's data dependences; the specific labeling strategies for characterizing def-order dependence and the carriers of a dependence; and the emphasis on showing that $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ have equivalent definitions of dependence. Earlier studies that use labels to analyze program behavior typically assume, without proof, that an instrumented characterization of program behavior can be used to reason about a language's implementation semantics. This assumption is problematic, since the typical goal of program analysis is to characterize a program's *actual* behavior. To rephrase this statement in the context of the current chapter, the observation that $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{M}_{\mathcal{H}}$ are distinct—albeit related—semantics implies that a characterization of a program's behavior w.r.t. $\mathbf{MI}_{\mathcal{H}}$ does not necessarily hold for $\mathbf{M}_{\mathcal{H}}$. One would like to be sure (*e.g.*) that *the use of labels does not impose any hidden implementation commitments on* $\mathbf{M}_{\mathcal{H}}$ *that limit the applicability of the analysis.* This concern is addressed by Lemma 4.2, which asserts that the definitions of flow dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ are equivalent.

4.1. Using Labels to Characterize Dependence

Various authors, including Cousot and Cousot [Cou80] and Nielson [Nie81, Nie87], distinguish between two kinds of program analyses. The first, the ***history-insensitive*** or ***first-order*** analysis, characterizes the *set* of states that a program generates. The second, the ***history-sensitive*** or ***second-order*** analysis, characterizes the *sequences* of states that a program generates. This distinction is interesting because the two

kinds of analyses characterize different aspects of program execution. First-order analyses typically generate assertions about the values that a program might compute. Examples of such assertions include "variable x always has the value 3 at point p", and "function f's second parameter is never $\perp$". Second-order analyses typically generate assertions about *how* a program computes its values. Examples of such assertions include "point p evaluates ten times before the evaluation of point q", and "program P exhibits the dependence $p \longrightarrow_d q$". This distinction is also interesting for pragmatic reasons. A semantics that simply characterizes a computation's current state is not a good starting point for analyzing second-order behavior. A more useful starting point for such an analysis is a *history-sensitive semantics—i.e.*, a semantics that gives a computation's current state s, *as well as* other information about states that precede (or succeed) s.

The semantics given in Chapter 2 is *not* a history-sensitive semantics, in the sense of [Nie81]. The balance of this chapter develops such semantics for language $\mathcal{H}$, and uses them to analyze a program's dependences.

One approach to developing a second-order semantics treats a program's statement-evaluation function as a map from a *sequence* of states to a *sequence* of states [Cou80, Nie81]. The resulting ***trace semantics*** generates an output that gives (1) a computation c's final state, and (2) a complete record of c's intermediate states. Figure 4.1 shows a trace of an example program P's evaluation w.r.t. an empty initial store. The depicted trace, trace t, is the trace that reaches statement [4]. The dependences incident on [4] can be computed by using t to unravel P's execution. Specifically, statement [4] manipulates four objects in the store: the global environment, reference y, structure *s1*, and reference *intp*. An inspection of t shows that these objects were created by the initial program point and statements 3, 1, and 2, respectively. Program P therefore exhibits $\textbf{initial}_1 \longrightarrow_f [4]$, $[1] \longrightarrow_f [4]$, $[3] \longrightarrow_f [4]$, and $[2] \longrightarrow_o [4]$.

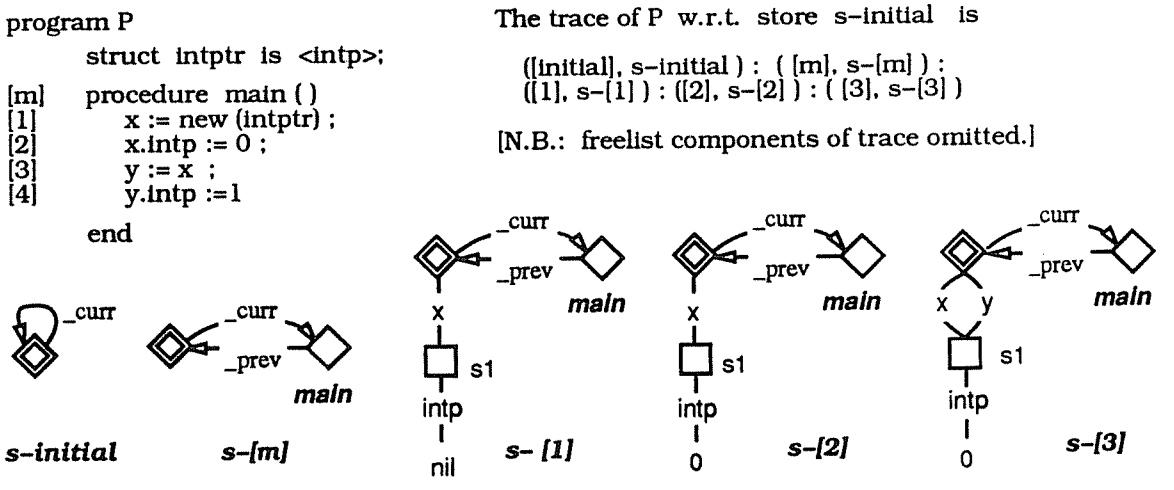


Figure 4.1. A trace of an example computation.

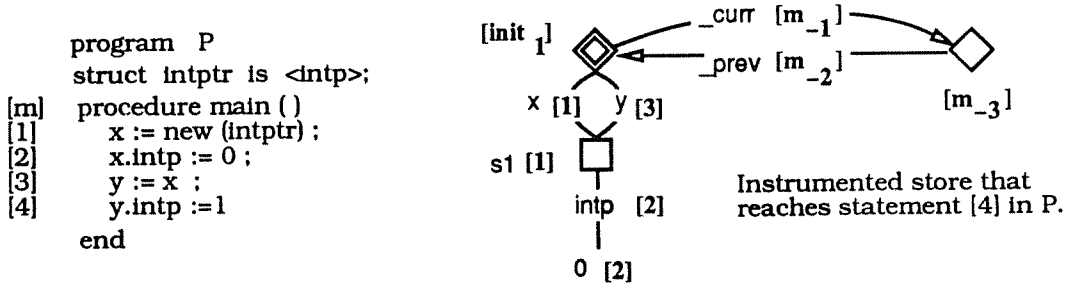


Figure 4.2. A state in an instrumented computation. Bold, bracketed strings are defining-point labels. $\mathbf{init}_1$ is short for $\mathbf{initial}_1$, a program's initial program point. $\mathbf{m}_{-3}$, $\mathbf{m}_{-2}$, and $\mathbf{m}_{-1}$ are the three program points that initialize procedure *main* ( )'s local environment.

A second approach to defining a history-sensitive semantics treats a statement as map from a *labeled state* to a *labeled state*. These labels give facts about a computation's history. A semantics that labels every object with its defining program point is illustrated in Figure 4.2. This semantics, referred to here as an ***instrumented semantics***, is a straightforward extension of the semantics given in Chapter 2. The program depicted in Figure 4.2, program P, is the one depicted in Figure 4.1. The store depicted in Figure 4.2, store s, is the ***instrumented store*** that reaches statement [4]. The dependences incident on [4] can be computed by using s to discover facts about P's execution. Specifically, statement [4] manipulates four objects in s: the global environment, reference y, structure $s1$, and reference *intp*. An inspection of s's labels shows that these objects were created by the initial program point and statements 3, 1, and 2, respectively. Program P therefore exhibits $\mathbf{initial}_1 \longrightarrow_f [4]$, $[1] \longrightarrow_f [4]$, $[3] \longrightarrow_f [4]$, and $[2] \longrightarrow_o [4]$.

The program depicted in Figure 4.1 and 4.2 exhibits the same flow and output dependences w.r.t the two semantics. The principal reason for preferring the instrumented semantics is that it yields a more efficient characterization of a program's evaluation. As Figures 4.1 and 4.2 illustrate, labels condense a computation's execution history, and make facts about computations easier to retrieve.

What follows now is an informal description of a set of instrumented semantics that yield exact, albeit possibly uncomputable, characterizations of a computation's data dependences. It is important to keep in mind that the following definitions are not the same as those given in Chapter 3.

Flow dependences. Flow dependences are defined as *write-before-read* dependences (*cf.* §3.2.2). The instrumented semantics for characterizing flow dependence therefore pairs every non-atomic object *obj* with the name of the statement that writes *obj*. The instrumented characterization of flow dependence states that a computation exhibits $p \longrightarrow_f q$ when point q reads an object *obj* whose label equals p.

Atoms are not labeled because they are never shared; there is never more than one pointer to a given atom throughout a computation. Furthermore, the definition of $\mathcal{H}$ guarantees that:

- An atom a and a reference r to a are defined at the same point in a computation; and

- Atom a is never accessed without a corresponding access of r.

The dependences that arise from the read of an atom a are therefore the same as the dependences that arise from the read of its corresponding reference r.

Input dependences. Input dependences are defined as *read-before-read* dependences (*cf.* §3.2.2). The instrumented semantics for characterizing input dependence therefore pairs every non-atomic object *obj* with the names of all statements that read *obj*. (*N.B.*: this label is initially empty.) The instrumented characterization of input dependence states that a computation exhibits $p \longrightarrow_i q$ when point q reads an object *obj* whose label contains p.

Output dependences. Output dependences are defined as *write-before-write* dependences (*cf.* §3.2.2). The instrumented semantics for characterizing output dependence therefore pairs every reference *ref* with the name of the statement that writes *ref*. The instrumented characterization of output dependence states that a computation exhibits $p \longrightarrow_o q$ when point q overwrites a reference whose label equals p.

Structures are not labeled because output and anti-dependences do not arise through structures. $\mathbf{MI}_{\mathcal{H}}$, like $\mathbf{M}_{\mathcal{H}}$, does not overwrite structures.

Anti-dependences. Anti-dependences are defined as *read-before-write* dependences (*cf.* §3.2.2). The instrumented semantics for characterizing anti-dependence therefore pairs every reference *ref* with the names of all statements that read *ref*. (*N.B.*: this label is initially empty.) The instrumented characterization of anti-dependence states that a computation exhibits $p \longrightarrow_a q$ when point q overwrites a reference whose label contains p.

Def-order dependences. A def-order dependence is defined as a pair of flow dependences that arise at a specific field of a specific structure (*cf.* §3.2.2). The instrumented semantics for characterizing def-order dependence therefore pairs every reference *ref* of type t at structure s with *two* labels. The first label, which names *ref*'s defining point, is used to determine flow dependences. The second label on *ref*, the *prior-dependences label*, names those dependences $p \longrightarrow_f r$ that arise through reads of references of type t at s. (When a reference *ref* of type t at structure s is replaced with a reference ref_{new}, reference ref_{new} inherits *ref*'s prior-dependences label.) The instrumented characterization of def-order dependence states that a computation exhibits $p \longrightarrow_{do(r)} q$ when:

1. p precedes q in a program's abstract syntax tree; and either
2a. r reads a reference *ref* whose writing-point label is q, and whose prior-dependences label contains $p \longrightarrow_f r$; or
2b. r reads a reference *ref* whose writing-point label is p, and whose prior-dependences label contains $q \longrightarrow_f r$.

Carriers of a dependence. The instrumented semantics for characterizing a dependence's carriers maintains a computation's current occurrence string (*cf.* §3.3) as a part of that computation's state. Objects that are read and written at a point p are labeled with the current occurrence of p. When a data dependence $p \longrightarrow_d q$ is created by an operation on memory, the occurrence of q is checked against the occurrence of p to determine the carriers of $p \longrightarrow_d q$.

4.2. An Instrumented Semantics for Characterizing Flow Dependence

The current section uses an example instrumented semantics, $\mathbf{MI}_{\mathcal{H}}$, to illustrate the labeling techniques given.in the previous section. Semantics $\mathbf{MI}_{\mathcal{H}}$, whose definition is given in Appendix 2, labels every reference and non-atomic structure *obj* with the occurrence of the program point that creates *obj*. The evaluation of a program's initial program point, $\mathbf{initial}_1$, labels every non-atomic object in the initial store with the value $\mathbf{initial}_1$; a subsequent statement p that creates a non-atomic object *obj* labels *obj* with the *current occurrence* of p. $\mathbf{MI}_{\mathcal{H}}$ also maintains a computation c's current occurrence string as a part of c's state. A transition into the body of a loop l adds an "l" to the end of the current occurrence string. An exit from l strips all l's from the end of the string. A procedure call at point s saves the current occurrence string in the current procedure's local environment, then appends an "s" to the occurrence string. A return from a procedure restores the caller's occurrence string.

To show that $\mathbf{MI}_{\mathcal{H}}$ can be used to identify a program's dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$, it must be shown that the new definitions of dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ are equivalent to the definitions given in Chapter 3. This is demonstrated by first formalizing the definition of flow dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$, and then showing that an arbitrary computation exhibits the same set of flow dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$. Similar arguments justify the use of the other labeling techniques described in Section 4.1.

DEFINITION (***write of a memory object w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). A structure or reference is *written* (w.r.t. $\mathbf{MI}_{\mathcal{H}}$) when it is added to a store. □

DEFINITION (***write of a memory object at a state w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). Let p be a program point, σ a store, fl a freelist, *label* a label function, and *occ* an occurrence string. A structure (reference) *obj* is *written at state* (p, σ, fl, *label*, *occ*) (w.r.t. $\mathbf{MI}_{\mathcal{H}}$) iff the evaluation of p w.r.t. σ, fl, *label*, and *occ* writes *obj*. □

DEFINITION (***read of a memory object w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). A structure or reference is *read* (w.r.t. $\mathbf{MI}_{\mathcal{H}}$) when it is accessed by the evaluation of an identifier expression. □

DEFINITION (***read of a memory object at a state w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). Let p be a program point, σ a store, fl a freelist, *label* a label function, and *occ* an occurrence string. A structure (reference) *obj* is *read at state* (p, σ, fl) (w.r.t. $\mathbf{MI}_{\mathcal{H}}$) iff the evaluation of p w.r.t. σ, fl, *label*, and *occ* reads *obj*. □

The remarks in Chapter 3 about the circumstances under which objects are read and written also apply to the instrumented semantics. The definitions of $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ are identical, up to that part of $\mathbf{MI}_{\mathcal{H}}$'s definition that maintains labels and occurrence strings.

DEFINITION (***state transition relation for*** $\mathbf{MI}_{\mathcal{H}}$). The instrumented semantics' *state transition relation* $\cdots \vdash_I \cdots \longrightarrow \cdots$ is defined as follows:

$$prog \vdash_I state_i \longrightarrow^0 state_j \;\Leftrightarrow\; state_j = state_i$$

$$prog \vdash_I state_i \longrightarrow^n state_j \;\Leftrightarrow\; \exists\, state' : prog \vdash_I state_i \longrightarrow^{n-1} state' \;\wedge\; state_j = evalPt_I(prog, state')$$

$$prog \vdash_I state_i \longrightarrow^* state_j \;\Leftrightarrow\; \exists\, n : prog \vdash_I state_i \longrightarrow^n state_j$$

$$prog \vdash_I state_i \longrightarrow^+ state_j \;\Leftrightarrow\; \exists\, n > 0 : prog \vdash_I state_i \longrightarrow^n state_j$$

$$prog \vdash_I state_n \longrightarrow \cdots \longrightarrow state_m \;\Leftrightarrow\; \forall\, i : n \le i \le m-1 : prog \vdash_I state_i \longrightarrow^1 state_{i+1}$$ □

The expression $evalPt_I(prog, state')$ constitutes a minor abuse of notation. The function $evalPt_I$ (*cf.* Appendix 2) actually takes one formal parameter—a state—and three non-local parameters that

describe *prog*'s control-flow graph, structure declarations, and local identifiers.

(RE)DEFINITION (***true for all states between ...***). A predicate $P : State \rightarrow Bool$ is *true for all states between* $state_n$ *and* $state_m$ iff $prog \vdash_I state_n \longrightarrow \cdots \longrightarrow state_m$ implies that $P(state_i) = true$ for all $i : n < i < m$. □

DEFINITION (***trace of a computation w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). Let *prog* be a program and σ a store. The *trace* of *prog* on σ is the sequence $(p_1, \sigma_1, fl_1, label_1, occ_1) \cdots (p_n, \sigma_n, fl_n, label_n, occ_n), \cdots$, where $p_1 = \mathbf{initial}_1$; $\sigma_1 = \sigma$; fl_1 is the freelist that $\mathbf{MI}_{\mathcal{H}}$ pairs with σ; $label_1$ is the label function that pairs every object with the special value *undefined*; $occ_1 = \varepsilon$, the empty occurrence string; and, for all i, $prog \vdash_I (p_i, \sigma_i, fl_i, label_i, occ_i) \longrightarrow (p_{i+1}, \sigma_{i+1}, fl_{i+1}, label_{i+1}, occ_{i+1})$. □

DEFINITION (***occurrence-specific flow dependence w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). Let *prog* be a program with points p and q, and *InSet* a set of stores. Let *op* and *oq*, where $oq = oq_1 \cdots oq_{n-1}q$, be occurrences of p and q, respectively. Program *prog* exhibits a *flow dependence* $op \longrightarrow_f oq$ w.r.t. *InSet* iff $q \neq \mathbf{initial}_1$ and there exists a store $\sigma \in InSet$, a freelist fl, an instrumented state $(q, \sigma_q, fl_q, label_q, oq_1 \cdots oq_{n-1})$, and an object *obj* such that

- $prog \vdash_I (\mathbf{initial}_1, \sigma, fl, label, \varepsilon) \longrightarrow^+ (q, \sigma_q, fl_q, label_q, oq_1 \cdots oq_{n-1})$, where ε denotes the empty occurrence string, and *label* the label function that pairs every object with the label *undefined*;
- *obj* is read at $(q, \sigma_q, fl_q, label_q, oq_1 \cdots oq_{n-1})$, and
- *obj* is labeled *op*. □

DEFINITION. (***flow dependence w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). Let *prog* be a program with points p and q, and *InSet* a set of stores. Program *prog* exhibits a *flow dependence* $p \longrightarrow_f q$ w.r.t. *InSet* iff there exist occurrences of p and q, *op* and *oq*, such that *prog* exhibits $op \longrightarrow_f oq$ w.r.t. *InSet*. □

DEFINITION (***carriers of a dependence w.r.t.*** $\mathbf{MI}_{\mathcal{H}}$). The *carriers* of a dependence $p \longrightarrow_d q$ are the set of all *carrier* $([i, p],[j, q])$ (*cf.* §3.3) such that $[i, p] \longrightarrow_d [j, q]$ w.r.t. $\mathbf{MI}_{\mathcal{H}}$. □

This completes the formal definition of flow dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$. The following two lemmas show the equivalence of the definitions of flow dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$.

DEFINITION (***congruent stores***). Let $\sigma \in Store$ be a store, and $\sigma_I \in Store_I$ an instrumented store. Stores σ and σ_I are *congruent* iff σ can be obtained from σ_I by replacing all values in σ_I of the form (*return–pt*, *return–occstr*) (*i.e.*, all instrumented calling contexts) with values of the form *return–pt*. □

LEMMA 4.1. Let P be a program, and σ a store. Let $t = (p_1, \sigma_1, fl_1) \cdots (p_n, \sigma_n, fl_n) \cdots$ be the trace of $\mathbf{M}_{\mathcal{H}}(P, \sigma)$. Let $t_I = (p'_1, \sigma'_1, fl'_1, label'_1, occ'_1) \cdots (p'_n, \sigma'_n, fl'_n, label'_n, occ'_n) \cdots$, where $fl'_1 = fl_1$, be the trace of $\mathbf{MI}_{\mathcal{H}}(P, \sigma)$. Then, for all i:

1. $p'_i = p_i$;
2. σ'_i is congruent to σ_i;
3. $evalPt(p_i, \sigma_i, fl_i)$ and $evalPt_I(p'_i, \sigma'_i, fl'_i, label'_i, occ'_i)$ read the same objects;
4. occ_i is the occurrence string for $(p_1, \sigma_1, fl_1) \cdots (p_i, \sigma_i, fl_i)$; and

5. if $i > 1$ and *obj* is a non-atomic object in σ'_i, then $label'_i(obj)$ identifies the occurrence of that point that created *obj* in t.

PROOF (*sketch*). Assertions 1 and 2 are proved by induction on the length of t and t_I. Inuitively, these assertions are true because $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ have identical definitions, up to those parts of $\mathbf{MI}_{\mathcal{H}}$ that are concerned with maintaining labels and the current occurrence string.

Assertion 3 follows immediately from the definitions of *evalPt* and $evalPt_I$, which are identical up to that part of $evalPt_I$ that maintains labels.

The proof of assertion 4 is tantamount to showing that $\mathbf{MI}_{\mathcal{H}}$ incrementally computes a computation's occurrence string. This assertion is also proved by induction on the length of t and t_I. The induction hypothesis states that occ'_j, the occurrence string at the jth state in the trace of $\mathbf{MI}_{\mathcal{H}}(P, \sigma)$, is the occurrence string for $(p_1, \sigma_1, fl_1) \cdots (p_j, \sigma_j, fl_j)$. The induction hypothesis also characterizes how environments and (*return–pt, return–occurrence–string*) pairs are configured in σ'_j. Roughly speaking, the induction hypothesis asserts that executing k **return** instructions after the jth step in an instrumented computation would maintain the proper occurrence string, so long as there are no more than k active procedure calls at step j.

Assertion 5 is also proved by an induction on the number of states in t.

If $i = 2$, then the all objects in σ_1 are created at $\mathbf{initial}_1$. By the definition of $\mathbf{MI}_{\mathcal{H}}$, all accessible non-atomic objects in σ'_2 are labeled $\mathbf{initial}_1$.

Assume that the assertion holds for $i-1$. Then the induction hypothesis follows from the definitions of $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$. Specifically, the two semantics' versions of *evalPt* create the same sets of structures and references. Furthermore, $\mathbf{MI}_{\mathcal{H}}$ leaves all labels on all other objects in σ'_{i-1} unchanged.

□

LEMMA 4.2. Let P be a program, σ a store, and $d = p \longrightarrow_f q$ a dependence. Program P exhibits d w.r.t. σ according to the standard semantics (*i.e.*, $\mathbf{M}_{\mathcal{H}}$) iff P exhibits d w.r.t. σ according to the instrumented semantics (*i.e.*, $\mathbf{MI}_{\mathcal{H}}$). Furthermore, d is carried by (x, y) w.r.t. σ according to the standard semantics iff d is carried by (x, y) w.r.t. σ according to $\mathbf{MI}_{\mathcal{H}}$.

PROOF (*sketch*). Let o_p and o_q be occurrence strings, and $d' = [o_p, p] \longrightarrow_f [o_q, q]$ an occurrence-specific dependence. Lemma 4.2 follows from the claim that P exhibits d' w.r.t. σ according to the standard semantics iff P exhibits d' w.r.t. σ according to the instrumented semantics. The proof of this claim breaks down into two cases.

Standard implies instrumented. Let $t = (p_1, \sigma_1, fl_1) \cdots$, where $\sigma_1 = \sigma$, be the trace of $\mathbf{M}_{\mathcal{H}}(prog, \sigma)$. By the definition of flow dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$, t contains two states (p_j, σ_j, fl_j) and (p_k, σ_k, fl_k) such that:

- $p_j = p$;
- o_p is the occurrence string for $(p_1, \sigma_1, fl_1) \cdots (p_j, \sigma_j, fl_j)$;
- an object *obj* is written at (p_j, σ_j, fl_j);
- *obj* is not overwritten at any states between (p_j, σ_j, fl_j) and (p_k, σ_k, fl_k);
- $k > j$;

- $p_k = q$;
- o_q is the occurrence string for $(p_1, \sigma_1, \mathit{fl}_1) \cdots (p_k, \sigma_k, \mathit{fl}_k)$; and
- *obj* is read at $(p_k, \sigma_k, \mathit{fl}_k)$.

By Lemma 4.1, the *k*th state in the trace of $\mathbf{MI}_{\mathcal{H}}(P, \sigma)$, state'_k, is a state of the form $(q, \sigma'_k, \mathit{fl}'_k, \mathit{label}'_k, o_q)$, where σ'_k is congruent to σ_k. Lemma 4.1 also implies that *obj* is read at state'_k. The proof now breaks into two cases, according to whether *obj* is an atom:

> If *obj* is not an atom, then by Lemma 4.1 *obj* is labeled $[o_p, p]$. Since $k > j$, $q \neq \mathbf{initial}_1$. The definition of an occurrence-specific flow dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ implies that P exhibits d' w.r.t. σ.
>
> If *obj* is an atom, then, by the definition of $\mathbf{MI}_{\mathcal{H}}$, the reference *ref* to *obj* in σ'_k must also be read at state'_k. By the definition of $\mathbf{M}_{\mathcal{H}}$, *ref* must also have been created at the *j*th state in *t*. By Lemma 4.1, *ref* must now be labeled $[o_p, p]$. Since $k > j$, $q \neq \mathbf{initial}_1$. The definition of an occurrence-specific flow dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ implies that P exhibits d' w.r.t. σ.

Instrumented implies standard. Let $t' = (p'_1, \sigma'_1, \mathit{fl}'_1, \mathit{label}'_1, \mathit{occ}'_1) \cdots$, where $\sigma'_1 = \sigma$, be the trace of $\mathbf{MI}_{\mathcal{H}}(\mathit{prog}, \sigma)$. By the definition of dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$, t' contains a state $\mathit{state}'_k = (p'_k, \sigma'_k, \mathit{fl}'_k, \mathit{label}'_k, \mathit{occ}'_k)$ such that:

- $p'_k = q$;
- $\mathit{occ}'_k = o_q$;
- a non-atomic object *obj* is read at state'_k; and
- *obj* is labeled $[o_p, p]$.

Let *j* be the unique state in t' such that $p'_j = p$ and $\mathit{occ}'_j = o_p$. (To assume there is no such state contradicts the hypothesis that there exists an *obj* labeled $[o_p, p]$; to assume there is more than one such state contradicts the assertion that occurrence strings uniquely name every occurrence of every program point in the course of a computation.) Since state'_k read an object labeled $[o_p, p]$, $j < k$.

Let $t = (p_1, \sigma_1, \mathit{fl}_1) \cdots$, where $\sigma_1 = \sigma$ and $\mathit{fl}_1 = \mathit{fl}'_1$, be the trace of $\mathbf{M}_{\mathcal{H}}(\mathit{prog}, \sigma)$. By Lemma 4.1,

- object *obj* is written at $(p_j, \sigma_j, \mathit{fl}_j)$;
- *obj* is not overwritten at any states between $(p_j, \sigma_j, \mathit{fl}_j)$ and $(p_k, \sigma_k, \mathit{fl}_k)$;
- *obj* is read at $(p_k, \sigma_k, \mathit{fl}_k)$.

The definition of an occurrence-specific flow dependence w.r.t. $\mathbf{M}_{\mathcal{H}}$ now implies that P exhibits d' w.r.t. σ. □

4.3. Relation to Previous Work

The material presented in Chapter 4 stresses two ideas:

- Program-point labels can be used to characterize a pointer program's dependences.
- The proposed labeling strategies are consistent with the implementation semantics for language $\mathcal{H}$ given in Chapter 2.

The current section discusses the relationship between these ideas and earlier work on program analysis.

Program-point labels are an old and oft-used tool in program analysis. The Courant Institute's SETL group was perhaps the first to use a labeled representation of the store to analyze a language with dynamically-created structures. Their work is described in Schwartz's report on the optimization of SETL

programs [Sch75]. This report describes labeling techniques for determining whether an object *obj* might be written at *p* and read at *q*; written at *p* and overwritten at *q*; or read at *p* and overwritten at *q*. Schwartz uses these techniques to determine when it is safe to replace occurrences of the SETL assignment statement, which typically makes a new copy of its left-hand argument, with more efficient operations that update the affected structures in place. An appendix of this paper sketches a second algorithm that estimates a program's *conflicts* (*cf.* §3.4.3); this algorithm assumes a pointer-arithmetic-free subset of PL/I.

Other analyses that use program-point labels to analyze pointer-program behavior (w.r.t. various techniques for representing stores) have been given by various authors. Roughly speaking, these reports can be classified according to whether labels are used to estimate object lifetimes [Rug87, Cha87, Rug88, Hed88, Har89, Deu90]; estimate a program's dependences [Har89, Lar89]; estimate the carriers or distances of a program's dependences [Lar89, Goh90, Bod90]; or limit the size of an approximate representation of the heap [Jon82, Hud87, Cha87, Nei88, Str88, Lar89, Deu90, Goh90, Str90, Cha90]. Only three of these reports [Str88, Har89, Deu90] observe that the correctness of a labeling technique needs to be demonstrated w.r.t. a language's implementation semantics. Only two of these three reports gives such a proof; the third [Str88] sketches, but does not develop, a semantics for program analysis.

The labeling technique presented here for determining def-order dependences is new. The technique's principal advantage lies in its referentially transparent characterization of def-order dependence. Previous techniques for computing def-order dependences compared the dependences that arise through *objects of the same name* in *pairs* of stores that reached *different occurrences* of a single witness point, *r*. The technique given here examines the labels on the *individual* objects in the *individual* stores that reach *individual* occurrences of *r*. This allows store approximation to proceed independently of how structures are named (*cf.* Chapter 6). A second advantage of the technique is that it allows states that have been checked for def-order dependences to be discarded—thereby saving space.

The use of labels to compute carriers was first proposed by Horwitz, Pfeiffer, and Reps in [Hor89a]. The labeling technique given here is a new version of this earlier technique that also supports procedures.

The question of whether an analysis characterizes a program's behavior also arises in the study of denotational semantics. The principal problem with using denotational definitions to assess program behavior is that these semantics do *not* characterize a computation's intermediate sequences of states: they merely define a correspondence between a program's inputs and its outputs. The following three approaches have been used to bridge the gap between a language's denotational and implementation semantics.

- Some authors accept the assertion that a denotational semantics characterizes a program's second-order behavior "on faith." Reports that use this approach often show that a labeling strategy is consistent with the original semantics [Mil76, Ple81].
- A second approach, discussed by Mulmuley [Mul87], uses the notion of full abstraction to associate a denotational definition with a canonical operational semantics for a language.
- A third approach augments a denotational semantics with assumptions about a program's evalution. Deutsch, for example, uses this approach in his work on functional programs [Deu90].

5. AN APPROXIMATION SEMANTICS FOR ANALYZING FLOW DEPENDENCE

> *The semantic analysis of a program is the determination of the conditions under which the executions of this program terminate, fail to terminate, or lead to an error ... The semantic analysis of a program should also allow the properties of objects manipulated by a program to be determined at every point in that program.*
> —P. Cousot; trans. from introduction to [Cou78]

This chapter describes $\mathbf{MA}_{\mathcal{H}}$, an *approximation semantics* for $\mathcal{H}$ that yields safe, computable characterizations of a program's flow dependences. The semantics given in the previous chapter, $\mathbf{MI}_{\mathcal{H}}$, cannot be used to compute a program's dependences w.r.t. infinite sets of stores and nonterminating computations. Semantics $\mathbf{MA}_{\mathcal{H}}$ overcomes these limitations through the use of *abstract objects*—special objects that represent infinite sets of values, stores, and occurrence strings. Similar extensions of the other labeling techniques described in Chapter 4 yield effective algorithms for determining other kinds of data dependences.

Chapter 5 is divided into four sections.

Section 5.1 defines $\mathbf{MA}_{\mathcal{H}}$, an approximate interpretation for language $\mathcal{H}$ that gives a terminating characterization of a program's execution. Semantics $\mathbf{MA}_{\mathcal{H}}$ is constructed from $\mathbf{MI}_{\mathcal{H}}$ by introducing abstract objects into $\mathbf{MI}_{\mathcal{H}}$'s domain. The rules for interpreting $\mathbf{MI}_{\mathcal{H}}$ are then extended to obtain a conservative interpretation of a pointer program's meaning.

Section 5.2 shows that $\mathbf{MA}_{\mathcal{H}}$ yields a safe approximation to a program's flow dependences. *Abstract Interpretation*, a framework for comparing fixpoints, is first used to show that $\mathbf{MA}_{\mathcal{H}}$ correctly estimates the set of states that a computation generates w.r.t. $\mathbf{MI}_{\mathcal{H}}$. This result is then used to argue that $\mathbf{MA}_{\mathcal{H}}$ correctly estimates the set of dependences that a computation generates w.r.t. $\mathbf{MI}_{\mathcal{H}}$. The proof is then completed by using the equivalence of $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{M}_{\mathcal{H}}$ (*cf.* Chapter 4) to show that $\mathbf{MA}_{\mathcal{H}}$ yields safe estimates of a program's flow dependences.

Section 5.3 uses an observation about the definition of $\mathbf{M}_{\mathcal{H}}$ to sharpen $\mathbf{MA}_{\mathcal{H}}$'s characterization of program execution. A given store-access expression cannot have more than one meaning at a specific moment in a program's interpretation, w.r.t. $\mathbf{M}_{\mathcal{H}}$. It is not possible, for example, for an expression such as "*y.intp* + *y.intp*", where *y.intp* is an integer, to evaluate to the value 3. This anomaly, which *does* arise in the approximate interpretation of pointer programs, can be avoided (in certain cases) by sharpening the interpretation of selector expressions.

Section 5.4 concludes by reviewing related work.

The principal contribution of Chapter 5 is a set of safe and flexible algorithms for estimating a program's data dependences. This flexibility is achieved by splitting the definition of $\mathbf{MA}_{\mathcal{H}}$ into two components: a set of semantic functions that interpret the meaning of $\mathcal{H}$'s operators, and a *stateset estimation function*, ∇. The operator ∇ ensures that analyses terminate by restricting the number of distinct states that the analysis can output. The exact definition of ∇, however, is left unspecified. The principal reason for this decision is that there does not appear to be a best heuristic for estimating pointer program behavior (*cf.* Chapter 6). A second contribution of this chapter is the adjustment to $\mathbf{MA}_{\mathcal{H}}$'s definition described in Section 5.3.

5.1. An Approximation Semantics for Dependence Computation

Appendix 3 defines a semantics, $\mathbf{MA}_{\mathcal{H}}$, that yields effective algorithms for estimating flow dependences. This new semantics avoids nontermination through the use of ***abstract objects***—objects that represent infinite sets of ***concrete*** (*i.e.*, "ordinary") objects in $\mathbf{MI}_{\mathcal{H}}$'s domain of definition. One such abstract object, $\top^{AT}$, represents the set of all atomic values. Another, the *summary structure*, represents an arbitrarily large set of structures. $\mathbf{MA}_{\mathcal{H}}$ uses abstract objects to create finite stores and states that represent infinite sets of stores and states. These special objects are then used to estimate how programs operate on infinite sets of states and stores.

The next two sections describe $\mathbf{MA}_{\mathcal{H}}$. Section 5.1.1 describes $\mathbf{MA}_{\mathcal{H}}$'s domain of states. Section 5.1.2 describes $\mathbf{MA}_{\mathcal{H}}$'s meaning function.

5.1.1. The domain of abstract states

Appendix 3 defines a semantics that gives conservative estimates of a program's flow dependences. The definition of flow dependence given in Chapter 4 suggests that this semantics, $\mathbf{MA}_{\mathcal{H}}$, must not underestimate—and may safely overestimate—the set of instrumented states that a program might compute. It is safe, in other words, for $\mathbf{MA}_{\mathcal{H}}$ to overestimate

- the set of program-point occurrences that a state might reach;
- the set of paths than a store might contain; and
- the set of program points that might have defined an object.

Semantics $\mathbf{MA}_{\mathcal{H}}$ uses *regular expressions* to abstract a computation's current occurrence string. An abstract occurrence string is a regular expression that denotes a *set* of occurrence strings. The abstract occurrence string $x(yz)^+$, for example, denotes the set of all occurrence strings of the form xW, where W is a string of one or more pairs of yz's.

Semantics $\mathbf{MA}_{\mathcal{H}}$ uses ***embeddings*** to abstract stores. An embedding, roughly speaking, is a value-preserving, path-preserving map from one store to a second. Figure 5.1 motivates the notion of an embedding, using stores from a program's standard interpretation. Store s_1, which has strictly fewer paths than store s_2, is trivially embeddable in s_2. Note that the example computation $P:s_2$ exhibits strictly more

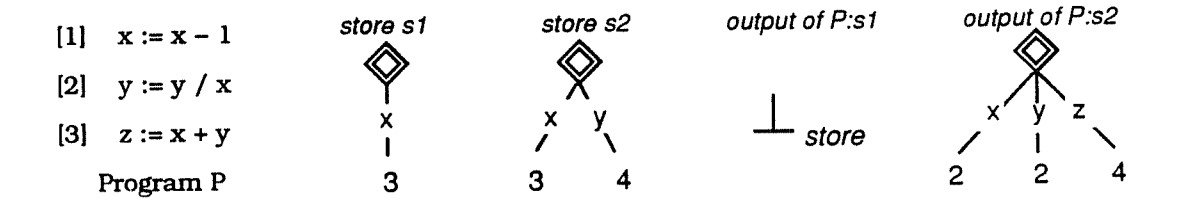
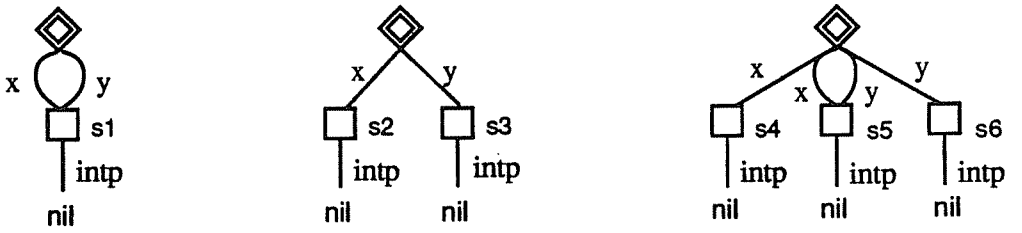


Figure 5.1. Store *s2* can be used in place of store *s1* to estimate a program's dependence, since *s2* contains strictly more paths than *s1*.

dependences than $P:s_1$, since the latter fails to terminate.

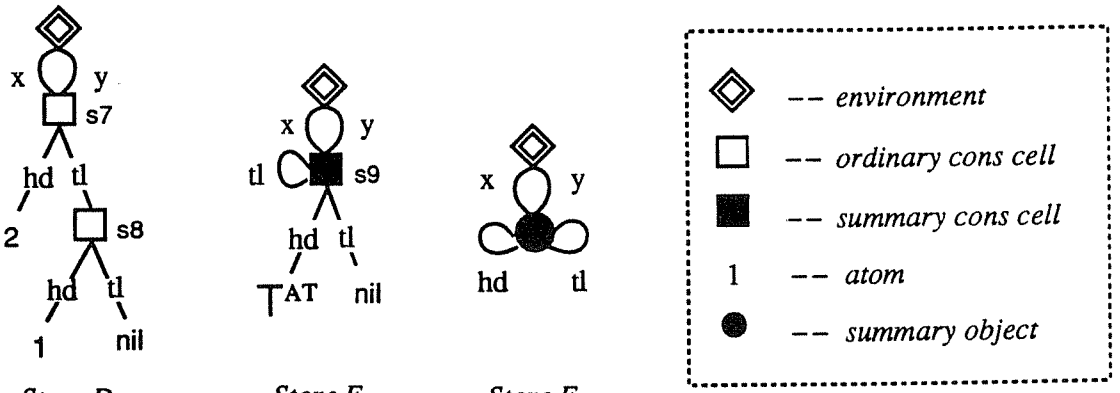
Figure 5.2 depicts embeddings of stores into *abstract* stores. The abstract stores depicted in Figure 5.2 differ from $\mathbf{MI}_{\mathcal{H}}$'s stores in the following four ways:

- Structures in abstract stores may have several edges with a given selector. This allows arbitrarily sets of structures to be condensed into a single, representative structure.
- Abstract atoms may have the value $\top^{AT}$.
- Structures in abstract stores may have more than one type.
- Abstract stores have two kinds of structures. White boxes in Figure 5.2 represent ***ordinary structures***: structures that are the image of no more than one structure in an embedding. Black boxes represent ***summary structures***: structures that may be the image of arbitrarily many structures in an



Store A *Store B* *Store C*

A and B embed in C. One embedding maps s1 to s5; the other maps s2 and s3 to s4 and s6.
A is not embeddable in B. s1 cannot be mapped onto both s2 and s3.
B is not embeddable in A. s1, an ordinary vertex, cannot be the image of s2 and s3.



Store D *Store E* *Store F*

D embeds in E. The embedding sends s7 and s8 to s9, and both integers to $\top^{AT}$.

D also embeds in F. The embedding sends every atom and cons cell in D onto F's special summary object. This object has type {atom,conscell} and value $\top^{AT}$.

Figure 5.2. Embeddings of stores in abstract stores.

embedding. Intuitively, a summary structure (and its incident edges) depicts the graph that results when an arbitrarily large subgraph of a store is condensed into a single, representative node.

Semantics $\mathbf{MA}_{\mathcal{H}}$ uses ***label-preserving*** embeddings to abstract instrumented stores. Intuitively, let o be a reference (or non-atomic structure) that is mapped to o_A in an abstract store. Assume that an occurrence op of a point p defined o. Then o_A's label must also assert that op might have defined o_A.

The assertion that one object is abstracted by a second is formalized as a relation, $\rhd$. The following is an informal definition of $\rhd$; a precise definition of this relation is given in Appendix 4.

DEFINITION (***occurrence-string abstraction***). An approximate occurrence string occ_A abstracts the occurrence string occ, written $occ \rhd occ_A$, iff occ is in the set of occurrence strings denoted by (the regular expression) occ_A. □

DEFINITION (***store abstraction***). Let $\sigma \in Store$ and $\sigma_A \in Store_A$ be stores. Store σ_A *abstracts* σ, written $\sigma \rhd \sigma_A$, iff there exists a map f such that

* f maps every accessible structure and reference in σ into σ_A.
* f maps σ's global environment to σ_A's global environment.
* ***f preserves kinds***: every ordinary structure in σ_A is the image of at most one structure in σ.
* ***f preserves types***: f maps every structure in σ of type t to a structure whose type includes t.
* ***f preserves atoms***: f maps every atom in σ of value v to an atom whose value is either v or $\top^{AT}$.
* ***f preserves contexts***: f maps every saved calling context in σ of the form $(pt, retocc)$ to a comparable context; *i.e.*, a context (pt, occ), where $retocc \rhd occ$.
* ***f preserves references***: if r is a reference to a structure s in σ, and r_A and s_A are the images of r and s under f, then r_A must reference s_A in σ_A. □

DEFINITION (***labeled store abstraction***). Let $ls = (\sigma, label) \in Store_I \times Label$ and $ls_A = (\sigma_A, label_A) \in Store_A \times Label_A$ be labeled stores. Labeled store ls_A *abstracts* ls, written $ls \rhd ls_A$, iff there exists a map f such that $\sigma \rhd \sigma_A$ by f, and ***f preserves labels***: that is, f maps every object in σ to a comparably labeled object in σ_A. □

DEFINITION (***state abstraction***). Let $state = (pt, \sigma, fl, occ, label) \in State_I$ and $state_A = (pt_A, \sigma_A, fl_A, occ_A, label_A) \in State_A$. State $state_A$ *abstracts* $state$, written $state \rhd state_A$, iff $pt = pt_A$, $occ \rhd occ_A$, and $(\sigma, label) \rhd (\sigma_A, label_A)$. □

DEFINITION (***pwr***). Let D be a set. The expression $pwr(D)$ denotes the powerset of D—the set of all subsets of D. □

DEFINITION (***stateset abstraction***). Let $state_* \in pwr(State_I)$ and $state_{*A} \in pwr(State_A)$. Stateset $state_{*A}$ *abstracts* $state_*$, written $state_* \rhd state_{*A}$, iff for all $state \in state_*$ there exists a $state_A \in state_{*A}$ such that $state \rhd state_A$. □

A similar set of ***subsumption relations*** can be defined between abstract objects. The store subsumption relation $\sqsubseteq$ is based on a kind-, type-, atom-, context-, and reference-preserving embedding of one abstract store into a second. A pair of example abstract stores that satisfies this relation is depicted in Figure 5.2. E $\sqsubseteq$ F in Figure 5.2 by a map that sends every atom and cons cell in E onto the special summary object in F. A precise definition of $\sqsubseteq$ is given in Appendix 4.

The $\sqsubseteq$ relation is a reflexive and transitive relation. The domain of abstract statesets can therefore be ordered by using $\sqsubseteq$ to partition $pwr(State_A)$ into equivalence classes. Define statesets $state_*$ and $state'_*$ to be equivalent w.r.t. $\sqsubseteq$, written $state_* \sim state'_*$, iff $state_* \sqsubseteq state'_*$ and $state'_* \sqsubseteq state_*$. Then the set of equivalence classes of $pwr(State_A)$ under $\sim$, $pwr(State_A) / \sim$, is a partial order. This subsumption-induced ordering is important for formalizing the notion of a safe approximation. Section 5.2 argues that dependence is monotonic w.r.t. orderings on $pwr(State_A)$ induced by $\sqsubseteq$. Section 5.2 also argues that $\mathbf{MA}_{\mathcal{H}}$ is monotonic w.r.t. this ordering. These two results justify this use of embeddings to estimate dependence.

5.1.2. An approximate interpretation for $\mathcal{H}$

Semantics $\mathbf{MA}_{\mathcal{H}}$ is a *nondeterministic* extension of $\mathbf{MI}_{\mathcal{H}}$ whose domain includes abstract and concrete objects. To ensure that $\mathbf{MA}_{\mathcal{H}}$ gives a conservative picture of a program's dependences w.r.t. an abstract store s, the semantics must estimate all possible interpretations of the stores that s abstracts.

The interpretation of assignment statements w.r.t. $\mathbf{MA}_{\mathcal{H}}$ is illustrated in Figures 5.3 and 5.4. Figure 5.3 shows the interpretation of assignment statements in the presence of nondeterminisim. The expression x w.r.t. s_0 denotes *three* objects: the atomic structure $\top^{AT}$; structure s_1; and structure s_2. The expression "$[p]$ *x.intp* := 0" could therefore have any of three meanings: the one shown in store s_1; the one shown in s_2, and $\perp$, the error element, if x is taken to be $\top^{AT}$. In semantics $\mathbf{MA}_{\mathcal{H}}$, the meaning of p w.r.t. s_0 is $\{s_1, s_2\}$. Both stores are included in the set because the definition of dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ (*cf.* §5.1.1) implies that it is safe to overestimate the set of states that a computation might generate. The error state is omitted from the result because computations that err terminate, yielding no further dependences.

Figure 5.4 shows the interpretation of assignment statements in the presence of summary structures. In the previous example, two references of type *intp* (at s_1 and s_2) were replaced with new references of type *intp* (to the integer 0). Both references were situated at ordinary structures. References that are situated at summary structures, on the other hand, are left untouched by assignment. Figure 5.4 illustrates why. The second store in Figure 5.4, s_1, abstracts the first, s_0. If the evaluation of "*x.tl.tl* := *y*" w.r.t. store s_1 removed the edge in s_1 labeled *tl*, then the updated s_1 would no longer abstract the updated s_0.

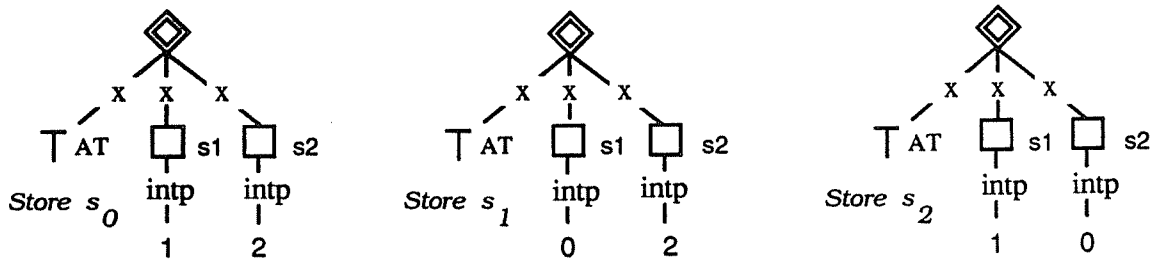


Figure 5.3. The effect of nondeterminism on store evaluation. Stores s_1 and s_2 represent two possible interpretations of "*x.intp* := 0" w.r.t. store s_0.

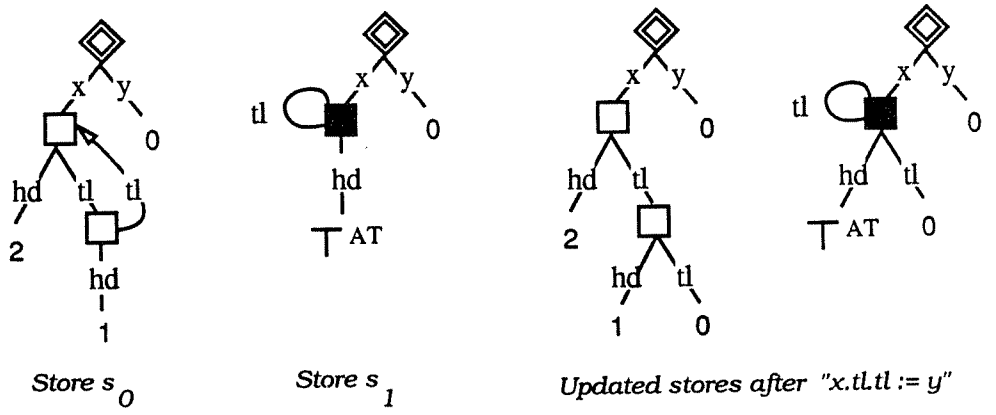


Figure 5.4. Why edge removal at summary vertices is unsafe. Store s_1 abstracts store s_0. *Not* replacing the *tl* edge at the summary structure ensures that s_1 continues to abstract the updated s_0.

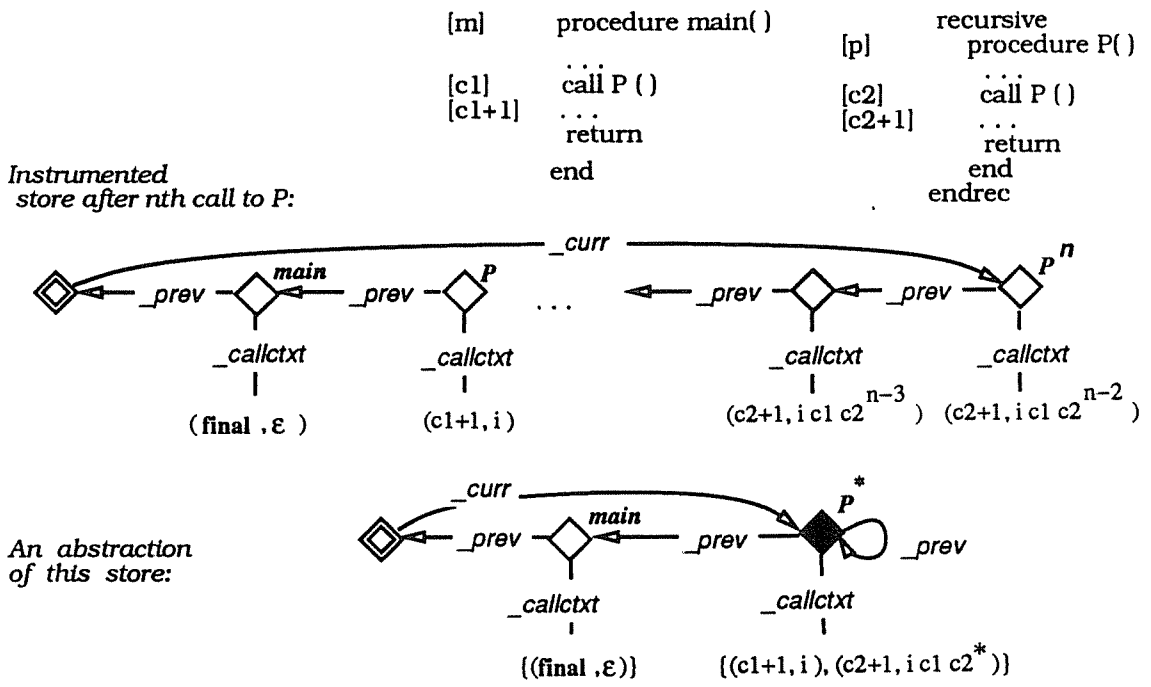
The ambiguous nature of abstract objects also affects the interpretation of predicates. Let store s_0, for example, be a store in which x references $\top^{AT}$. The meaning of "[p] **if** $x \stackrel{?}{=} 0$" w.r.t. s_0 is *both* true and false, since $\top^{AT}$ denotes zero and non-zero integers. An s_0 that reached p during an analysis would propagate to both of p's control-flow successors.

Figure 5.5 illustrates the effect of nondeterminism on the return statement. Objects pointed to by edges labeled *_callctxt* are calling contexts: (*return–point*, *return – occurrence –string*) pairs that record the state of the computation before a procedure call. When a return statement is evaluated w.r.t. the approximate store in Figure 5.5, the interpretation identifies *two* valid avenues of return: point [c2+1] in procedure P with environment P^* and occurrence string $\mathbf{initial}_2\, c1c2^*$, and point [c1+1] in procedure *main* with environment *main* and occurrence string $\mathbf{initial}_2$.[4]

The framework used to prove $\mathbf{MA}_{\mathcal{H}}$'s safety leads to a second important difference between the two semantics. $\mathbf{MA}_{\mathcal{H}}$ maps an initial store to a *set* of *states*. This set characterizes the calls to $evalPt_A$ that $\mathbf{MA}_{\mathcal{H}}$ makes over the course of a computation. Let $final_*$, for example, be the output of $\mathbf{MA}_{\mathcal{H}}$ when applied to an initial store σ, Assume, further, that $\mathbf{MA}_{\mathcal{H}}$, when run on σ, invokes $evalPt_A$ with an argument of the form $state = (pt, \sigma, fl, label, occ)$. Then $final_*$ either contains *state*, or a state that subsumes *state*.

A third difference between $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{MA}_{\mathcal{H}}$ is $\mathbf{MA}_{\mathcal{H}}$'s use of a stateset estimation operator, ∇, to ensure that analyses terminate. $\mathbf{MA}_{\mathcal{H}}$ assumes that ∇ is ***extensive***: *i.e.*, that ∇ maps every stateset s to a stateset

[4]Technically, there are four avenues of return: (1) [c1+1] with *main*, (2) [c1+1] with P^*, (3) [c2+1] with *main*, and (4) [c2+1] with P^*. Avenues (2) and (3), however, are not valid: (2), for example, pairs point [c1+1], which lies in *main*(), with a local environment defined by the procedure P. These invalid avenues of return could be eliminated by treating references as objects of type $Loc + Loc \times Context$. References labeled *_prev*, which would have type $Loc \times Context$, would be paired with the return point and the occurrence string of the calling procedure. Other references would have type Loc.



Return from nth call to P in approximation semantics (w.r.t. abstraction) yields two stores:

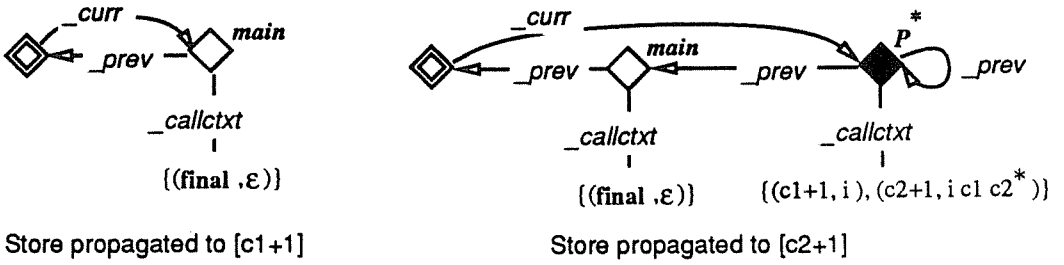


Figure 5.5. The evaluation of the return statement w.r.t. $\mathbf{MA}_{\mathcal{H}}$. The letter *i* in occurrence strings is a shorthand for $\mathbf{initial}_2$, the program point that first invokes *main*( ).

that subsumes *s*. $\mathbf{MA}_{\mathcal{H}}$ also assumes that ∇ limits the length of every ascending chain in $pwr(State_A)$. Intuitively, this implies that ∇ limits the number of different states that an analysis can generate. Such a limit can be imposed by restricting the size and content of approximate stores. One such ∇, depicted in Figures 5.6 and 5.7, maps updated atoms to $\top^{AT}$; uses one summary structure to represent all cons cells allocated at point [4]; and pairs every occurrence of a point *p* in the while loop at [3] with the approximate occurrence string $[3]^+p$. Other definitions of ∇ are explored in Chapter 6.

A final assumption about $\mathbf{MA}_{\mathcal{H}}$ is that its versions of $\mathbf{MI}_{\mathcal{H}}$'s primitive operators are monotonic w.r.t. the embedding-induced ordering on atoms. For example, $\mathbf{MA}_{\mathcal{H}}$'s version of + returns $\top^{AT}$ when either of its

arguments is $\top^{AT}$.

5.2. Proving the Correctness of the Approximate Interpretation

The following definitions characterize a program's flow dependences w.r.t. $\mathbf{MA}_{\mathcal{H}}$.

DEFINITION. (***occurrence-specific flow dependence w.r.t.*** $\mathbf{MA}_{\mathcal{H}}$). Let $prog \in Program_{\mathcal{H}}$, and *InSet* a set of stores. Let p and q be points in *prog*. Let *op* and *oq*, where $oq = oq_1 \cdots oq_{n-1} q$, be occurrences of p and q, respectively. Program *prog* exhibits a ***flow dependence*** $op \longrightarrow_f oq$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$ and *InSet* iff $q \neq \mathbf{initial}_1$, and there exists a $\sigma \in InSet$ and a $(q, \sigma', fl, label, occ) \in \mathbf{MA}_{\mathcal{H}}(prog, \sigma)$ such that

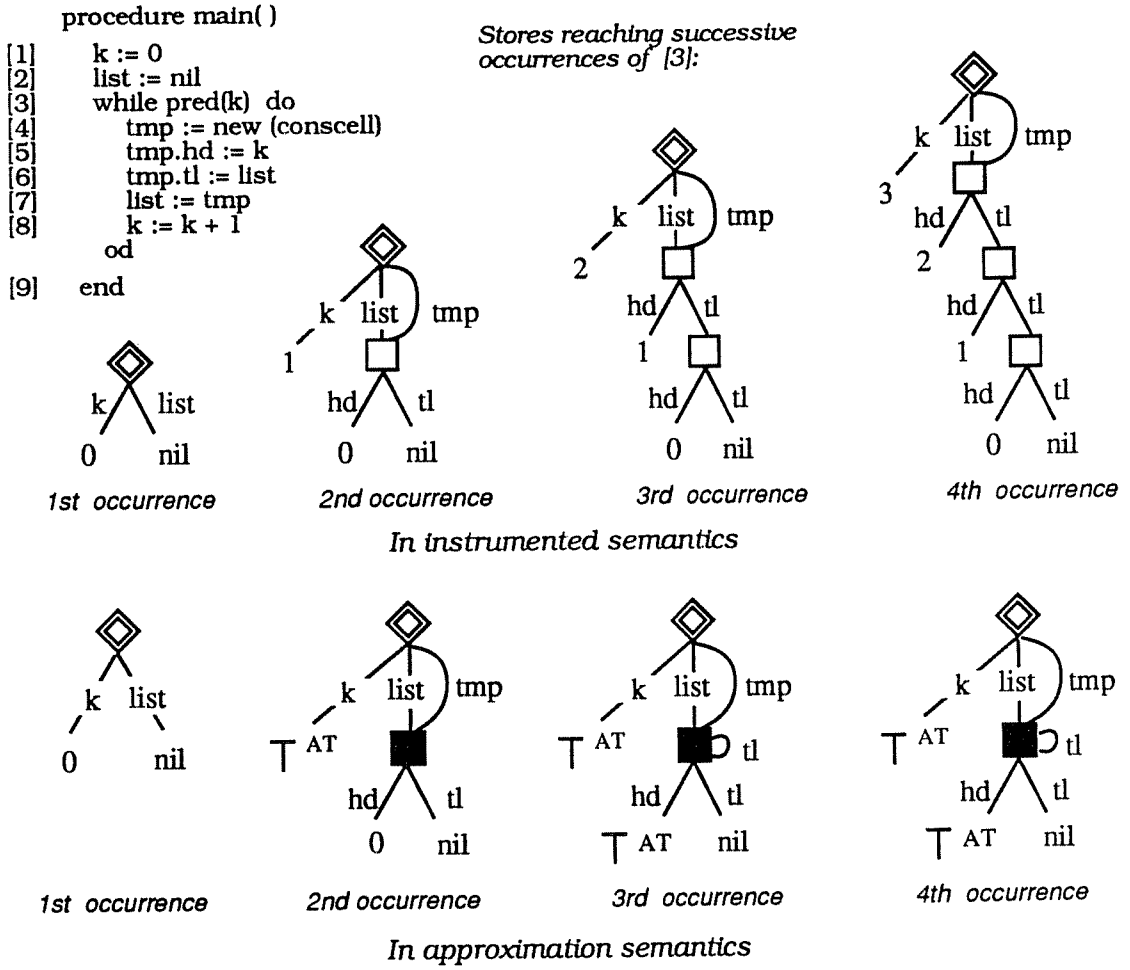


Figure 5.6. The progress of an example computation, relative to $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{MA}_{\mathcal{H}}$ and an empty initial store. The strategy used to limit the sizes of the approximate stores is described in the text. Labels and local environments are not shown.

$oq_1 \cdots oq_{n-1} \rhd occ$ and $evalPt_A((q, \sigma', fl, label, occ))$ accesses an object whose label abstracts *op*. □

DEFINITION (***flow dependence w.r.t.*** $\mathbf{MA}_{\mathcal{H}}$). Let $prog \in Program_{\mathcal{H}}$, and *InSet* a set of stores. Let *p* and *q* be points in *prog*. Program *prog* exhibits a ***flow dependence*** $p \longrightarrow_f q$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$ and *InSet* iff there exist occurrences of *p* and *q*, *op* and *oq*, such that *prog* exhibits $op \longrightarrow_f oq$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$ and *InSet*. □

DEFINITION (***carriers of a flow dependence w.r.t.*** $\mathbf{MA}_{\mathcal{H}}$). Let *carrier*([*x*, *p*],[*y*, *q*]) denote the carrier of the occurrence-specific dependence $[x, p] \longrightarrow_d [y, q]$ (*cf.* §3.3). The ***carriers*** of a dependence $p \longrightarrow_d q$ are the set of all *carrier*([*i*, *p*],[*j*, *q*]) such that $[i, p] \longrightarrow_d [j, q]$ w.r.t. $\mathbf{MS}_{\mathcal{H}}$. □

The balance of Section 5.2 demonstrates that these four definitions represent a safe estimate of a program's dependences w.r.t. $\mathbf{M}_{\mathcal{H}}$. Intuitively, it will be demonstrated that a program *P* that exhibits

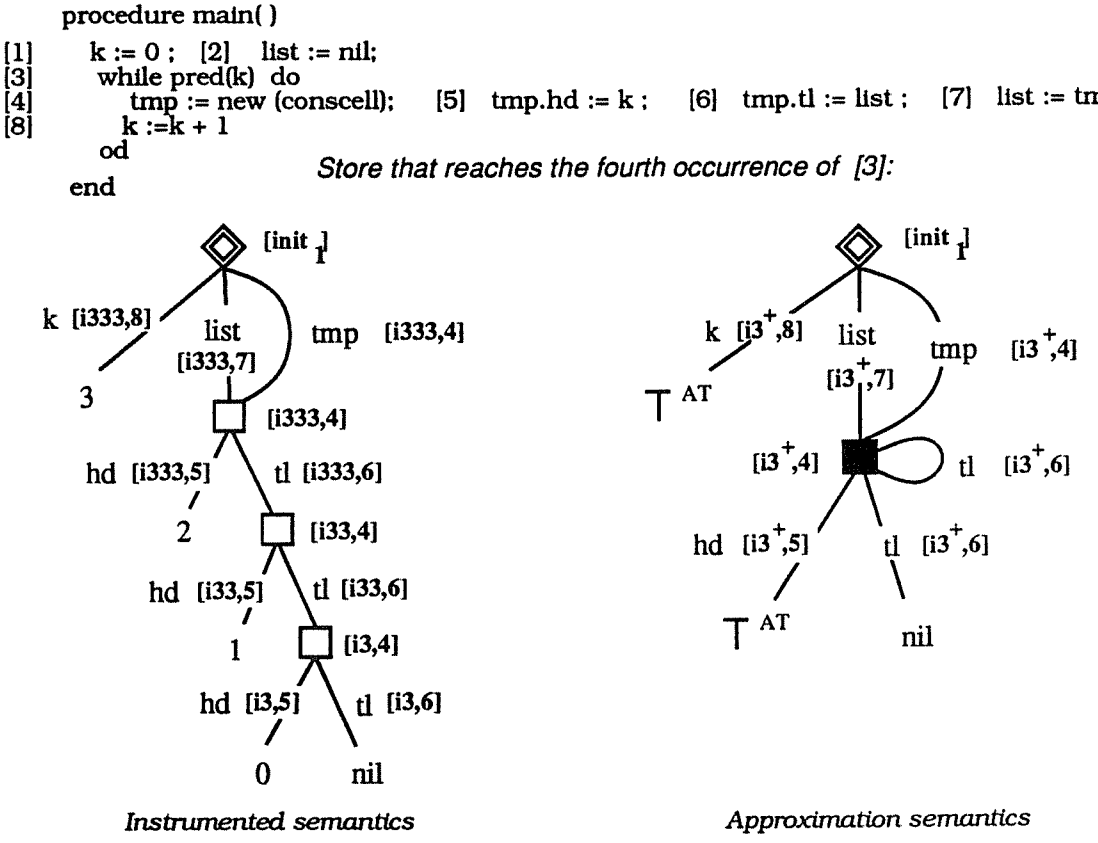


Figure 5.7. A "closeup" of two of the stores shown in Figure 5.7 that depicts the creation-point labels assigned to structures and references. Labels are shown in boldface. The symbol "I" is a shorthand for $\mathbf{initial}_2$, the program point that invokes *main*( ).

$op \longrightarrow_f oq$ w.r.t. $\mathbf{M}_{\mathcal{H}}$ and a set of states *InSet* must also exhibit $op \longrightarrow_f oq$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$ and any abstraction of *InSet*.

5.2.1. Abstract Interpretation

Abstract Interpretation is a well-known framework for program analysis developed by Cousot and Cousot [Cou77, Cou78, Abr87]. Abstract Interpretation simplifies the task of showing that the least fixpoint of a function f_C is approximated by the fixpoint of a related function f_A. This framework is typically used to show that an estimate of a program P's behavior obtained from some f_A is a safe estimate of an f_C that characterizes P's standard execution.

Abstract Interpretation is used below to demonstrate the safety of the definitions of flow dependence w.r.t. $\mathbf{MA}_{\mathcal{H}}$. The framework is first used to show that $\mathbf{MA}_{\mathcal{H}}$'s final output abstracts the set of all states generated by $\mathbf{MI}_{\mathcal{H}}$. It is then argued that the flow dependences created by the evaluation of a state w.r.t. $\mathbf{MI}_{\mathcal{H}}$ are a subset of those created by the evaluation of a comparable state w.r.t. $\mathbf{MA}_{\mathcal{H}}$. These two results establish that the flow dependences exhibited by a computation w.r.t. $\mathbf{MA}_{\mathcal{H}}$ are a superset of those exhibited w.r.t. a comparable instrumented computation. The safety of $\mathbf{MA}_{\mathcal{H}}$'s characterization of flow dependences then follows from the equivalence of $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{M}_{\mathcal{H}}$.

5.2.2. A static semantics for characterizing flow dependence

In the Cousots' formulation of abstract interpretation, the correctness of a semantics like $\mathbf{MI}_{\mathcal{H}}$ is demonstrated by using an intermediate semantics to pass from $\mathbf{MI}_{\mathcal{H}}$ to $\mathbf{MA}_{\mathcal{H}}$. This intermediate semantics, which the Cousots called a ***static semantics***, has a fixpoint that characterizes the set of states that are generated during the evaluation of $\mathbf{MI}_{\mathcal{H}}$.

Figure 5.8 gives a static semantics for $\mathbf{MI}_{\mathcal{H}}$, named $\mathbf{MS}_{\mathcal{H}}$. The definitions of $\mathbf{MI}_{\mathcal{H}}$ and $\mathbf{MS}_{\mathcal{H}}$ are identical, up to the definitions of $evalPgm_I$ and $evalPgm_S$. The static semantics' version of *evalPgm* simply "collects" into a set the states that are generated by the evaluation of $evalPgm_I$. This allows the definitions of dependence w.r.t. $\mathbf{MI}_{\mathcal{H}}$ to be rephrased in terms of a program's meaning, as follows:

DEFINITION (***occurrence-specific flow dependence w.r.t.*** $\mathbf{MS}_{\mathcal{H}}$). Let $prog \in Program_{\mathcal{H}}$, and *InSet* a set of stores. Let p and q be points in $Program_{\mathcal{H}}$. Let op and oq, where $oq = oq_1 \cdots oq_{n-1}q$, be occurrences of p and q, respectively. Program *prog* exhibits a *flow dependence* $op \longrightarrow_f oq$ w.r.t. *InSet* iff $q \neq \mathbf{initial}_1$, and there exists a $\sigma \in InSet$ and a $(q, \sigma', fl, label, oq_1 \cdots oq_{n-1}) \in \mathbf{MS}_{\mathcal{H}}(prog, \sigma)$ such that $evalPt_I((q, \sigma', fl, label, oq_1 \cdots oq_{n-1}))$ accesses an object labeled op. □

DEFINITION (***flow dependence w.r.t.*** $\mathbf{MS}_{\mathcal{H}}$). Let $prog \in Program_{\mathcal{H}}$, and *InSet* a set of stores. Let p and q be points in *prog*. Program *prog* exhibits a *flow dependence* $p \longrightarrow_f q$ w.r.t. $\mathbf{MS}_{\mathcal{H}}$ and *InSet* iff there exist occurrences of p and q, op and oq, such that *prog* exhibits $op \longrightarrow_f oq$ w.r.t. $\mathbf{MS}_{\mathcal{H}}$ and *InSet*. □

DEFINITION (***carriers of a flow dependence w.r.t.*** $\mathbf{MS}_{\mathcal{H}}$). The *carriers* of a dependence $p \longrightarrow_d q$ are the set of all *carrier* ([*i*, *p*],[*j*, *q*]) (*cf.* §3.3) such that $[i, p] \longrightarrow_d [j, q]$ w.r.t. $\mathbf{MS}_{\mathcal{H}}$. □

5.2.3. Relating $\mathbf{MS}_{\mathcal{H}}$ to $\mathbf{MA}_{\mathcal{H}}$

The claim that $\mathbf{MA}_{\mathcal{H}}$'s output abstracts $\mathbf{MS}_{\mathcal{H}}$'s output will be demonstrated by using Abstract Interpretation to compare the outputs of the semantics' state-transition loops, $evalPgm_S$ and $evalPgm_A$. In Cousot-style

$\mathbf{MS}_{\mathcal{H}}$: $Prog \rightarrow Store_I \rightarrow pwr(State_I)$

$\mathbf{MS}_{\mathcal{H}}(prog, \sigma) =$

```
let (structdecls, body) = prog in
  let body' = expand(initialize(body))
  and localIds = determineLocals(body')
  and structDecls = evalStructDecls(structdecls) in
  and fl = an arbitrarily large collection of locations not in σ
  and label = λ loc . (undefined, λ sel . undefined)
  and occ = ε, the empty occurrence string
  in
    let evalPgm_S = fix λ f . λ (state_*).
      let next_* = union_from state ∈ state_* :
                   if new's program-point component is final then { state }
                   elsif evalPt_I fails when evaluated on state then ∅
                   else evalPt_I (state)
                   fi
      in next_* ⊆ state_* → state_* [] f(next_*)
      end
    in evalPgm_S ({ (initial_1 , σ, fl, label, occ) })
end°
```

Figure 5.8. A static semantics for language $\mathcal{H}$. $evalPt_I$ denotes $\mathbf{MI}_{\mathcal{H}}$'s version of $evalPt$.

Abstract Interpretation, the claim that $evalPgm_A$'s output abstracts $evalPgm_S$'s output is demonstrated by proving the following assertions about $\mathbf{MA}_{\mathcal{H}}$ and $\mathbf{MS}_{\mathcal{H}}$.

1. The body of $evalPgm_S$ is continuous w.r.t. the subset ordering on $D_S = pwr(State_I)$, $evalPgm_S$'s domain of states. This ensures that $\mathbf{MS}_{\mathcal{H}}$ has a least fixpoint.
2. Function $evalPt_A$ is monotonic w.r.t. the subsumption ordering on $D_A = pwr(State_A)$, $evalPgm_A$'s domain of states. This ensures that $\mathbf{MA}_{\mathcal{H}}$ has fixpoints. (If $evalPgm_A$ is continuous, then iteration from $\perp$ computes $\mathbf{MA}_{\mathcal{H}}$'s least fixpoint; *cf.* [Cou77], Section 8.1.)
3. D_S and D_A are ***adjoined***; that is, there exist monotonic ***abstraction*** and ***concretization*** maps $abs : D_S \rightarrow D_A$ and $con : D_A \rightarrow D_S$ such that $state_* \subseteq con(abs(state_*))$ and $abs(con(state_{*A})) \sim state_{*A}$ for all $state_* \in D_S$ and $state_{*A} \in D_A$.
4. Functions $evalPt_I$ and $evalPt_A$ are ***congruent***; that is,

4a. For all $state_* \in D_S$, $abs(evalPt_I(state_*)) \sqsubseteq evalPt_A(abs(state_*))$; and

4b. For all $state_{*A} \in D_A$, $evalPt_I(con(state_*)) \subseteq con(evalPt_A(state_{*A}))$.

If $\mathbf{MA}_{\mathcal{H}}$, $\mathbf{MS}_{\mathcal{H}}$, *etc.* satisfy these requirements, then $abs(evalPgm_S(state_*)) \sqsubseteq evalPgm_A(abs(state_*))$, and $evalPgm_S(con(state_{*A})) \subseteq con(evalPgm_A(state_{*A}))$.

The Cousots show that *abs* and *con* determine one another: a suitable *abs* can be constructed from a monotonic *con*, and vice versa. The requisite relationship between $\mathbf{MA}_{\mathcal{H}}$ and $\mathbf{MS}_{\mathcal{H}}$ can therefore be established by defining a monotone concretization map from D_A to D_S and showing 1, 2, and 4b.

LEMMA 5.1 (***continuity of evalPgm$_S$***). *evalPgm$_S$* is continuous w.r.t. the subset ordering on D_S.

PROOF. Immediate from the definitions of continuity and *evalPgm$_S$*. □

LEMMA 5.2 (***monotonicity of evalPt$_A$***). *evalPt$_A$* is monotonic w.r.t. the subsumption ordering on D_A.

PROOF. Lemma 5.2 is proved with a series of lemmas that characterize its semantic functions. This proof, which is straightforward but rather long, is given in Appendix 5. □

DEFINITION. *con* : $D_A \rightarrow D_S$ maps a $state_{*A} \in D_A$ to { *state* ∈ *State$_I$* : { *state* } ▷ $state_{*A}$ }. □

LEMMA 5.3 (***monotonicity of con***). *con* is monotonic w.r.t. the subsumption ordering on D_A.

PROOF. By the definitions of ▷ and ⊑, $state_* \rhd state'_{*A}$ whenever $state_* \rhd state_{*A}$ and $state_{*A} \sqsubseteq state'_{*A}$. Thus, if $state_{*A} \sqsubseteq state'_{*A}$, then every { *state* } ∈ *State$_I$* that is abstracted by $state_{*A}$ is also abstracted by $state'_{*A}$. □

LEMMA 5.4 (***congruence of evalPt$_I$, evalPt$_A$***). For all $state_{*A} \in D_A$, $evalPt_I\,(con\,(state_{*A})) \subseteq con\,(evalPt_A\,(state_{*A}))$.

PROOF. This assertion is equivalent to the assertion that (*) $state \rhd state_A \Rightarrow evalPt_I\,(state) \rhd evalPt_A\,(state_A)$. The proof of (*), which resembles that of Lemma 5.2, is given in Appendix 6. □

LEMMA 5.5 (***congruence of evalPgm$_S$, evalPgm$_A$***). For all $state_{*A} \in D_A$, $evalPgm_S\,(con\,(state_{*A})) \subseteq con\,(evalPgm_A\,(state_{*A}))$.

PROOF. This claim follows from Lemmas 5.1, 5.2, 5.3, 5.4, and from the Cousot's extended framework for Abstract Interpretation, which supports ∇-like estimation operators—there called *widening operators* (*operateurs d'élargissement*). □

LEMMA 5.6 (***Abstract Interpretation Lemma***). If $\sigma \rhd \sigma_A$, then $\mathbf{MS}_{\mathcal{H}}(prog, \sigma) \rhd \mathbf{MA}_{\mathcal{H}}(prog, \sigma_A)$.

PROOF. Let $store_* = con\,(\sigma_A)$. Let

$state_* =$ **union_from** $\sigma' \in store_*$:
  union_from $fl \in Freelist$ such that fl is infinite and fl names no reachable structures in σ' :
    { (**initial**, σ', fl, *label*, ε) }

$state_{*A} =$ **union_from** $\sigma'_A \in store_{*A}$:
  union_from $fl \in Freelist$ such that fl is infinite and fl names no reachable structures in σ' :
    { (**initial**, σ'_A, fl_A, $label_A$, ε) }

By the definition of *con*, $state_* = con\,(state_{*A})$. It then follows, from Lemma 5.5, that (*) $evalPgm_S\,(state_*) \subseteq con\,(evalPgm_A\,(state_{*A}))$.

Observation (*) can now be used to show that $\mathbf{MS}_{\mathcal{H}}(prog, \sigma) \rhd \mathbf{MA}_{\mathcal{H}}(prog, \sigma_A)$. Let *stateset* = { (**initial**, σ, *fl*, *label*, ε) }. Clearly, *stateset* ⊆ $state_*$. By the continuity of $\mathbf{MS}_{\mathcal{H}}$ w.r.t. ⊆ (Lemma 5.1), (**) $evalPgm_S\,(stateset) \subseteq evalPgm_S\,(state_*)$. Since the subset relation is transitive, (*) and (**) imply that $evalPgm_S\,(stateset) \subseteq con\,(evalPgm_A\,(state_{*A}))$. Hence, by the definition of ▷, $\mathbf{MS}_{\mathcal{H}}(prog, \sigma) \rhd \mathbf{MA}_{\mathcal{H}}(prog, \sigma_A)$. □

5.2.4. Relating dependences w.r.t. $\mathbf{MA}_{\mathcal{H}}$ and $\mathbf{M}_{\mathcal{H}}$

THEOREM 5.1 (*safety of* $\mathbf{MA}_{\mathcal{H}}$ *w.r.t.* $\mathbf{MS}_{\mathcal{H}}$). Let *prog* be a program, and $\sigma \rhd \sigma_A$. Let *op* and *oq* be two occurrences of points in *prog*. Let $d = op \longrightarrow_f oq$. If *prog* exhibits *d* w.r.t. σ and $\mathbf{MS}_{\mathcal{H}}$, then *prog* exhibits *d* w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$.

PROOF. If program *prog* exhibits *d* w.r.t. $\mathbf{MS}_{\mathcal{H}}$, then $\mathbf{MS}_{\mathcal{H}}(prog, \sigma)$ contains a state, $(q, \sigma, fl, label, occ)$, such that $oq = append(occ, q)$ and the evaluation of an identifier expression *exp* at this state accesses an object labeled *op*. Lemma 5.6, however, implies that $\mathbf{MA}_{\mathcal{H}}(prog, \sigma_A)$ contains a state that abstracts $(q, \sigma, fl, label, occ)$. The second corollary to Lemma A.3 in Appendix 6, which characterizes the approximation semantics' evaluation of identifier expressions, asserts that the evaluation of *exp* at this state accesses an object whose label abstracts *op*. The definition of dependence w.r.t. $\mathbf{MA}_{\mathcal{H}}$ now implies that program *P* exhibits $op \longrightarrow_f oq$ w.r.t. σ. □

COROLLARY 1. Let *prog* be a program, and $\sigma \rhd \sigma_A$. Let *p* and *q* be two points in *prog*. If *prog* exhibits $p \longrightarrow_f q$ w.r.t. σ and $\mathbf{MS}_{\mathcal{H}}$, then *prog* exhibits $p \longrightarrow_f q$ w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$.

PROOF. Immediate from the definitions of flow dependence w.r.t. $\mathbf{MS}_{\mathcal{H}}$ and $\mathbf{MA}_{\mathcal{H}}$. □

COROLLARY 2. Let *prog* be a program, and $\sigma \rhd \sigma_A$. Let *p* and *q* be two points in *prog*. Let *prog* exhibit $d = p \longrightarrow_f q$ w.r.t. σ and $\mathbf{MS}_{\mathcal{H}}$. If *d* is carried by (x, y) w.r.t. σ and $\mathbf{MS}_{\mathcal{H}}$, then *d* is carried by (x, y) w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$.

PROOF. Immediate from the definitions of a dependence's carriers w.r.t. $\mathbf{MS}_{\mathcal{H}}$ and $\mathbf{MA}_{\mathcal{H}}$. □

THEOREM 5.2 (*safety of* $\mathbf{MA}_{\mathcal{H}}$ *w.r.t.* $\mathbf{M}_{\mathcal{H}}$). Let *prog* be a program, and $\sigma \rhd \sigma_A$. Let *op* and *oq* be two occurrences of points in *prog*. If *prog* exhibits $op \longrightarrow_f oq$ w.r.t. σ and $\mathbf{M}_{\mathcal{H}}$, then *prog* exhibits $op \longrightarrow_f oq$ w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$.

PROOF.

(1)	$op \longrightarrow_f oq$ w.r.t. $\mathbf{M}_{\mathcal{H}}$ iff $op \longrightarrow_f oq$ w.r.t. $\mathbf{MI}_{\mathcal{H}}$	(Lemma 4.2)
(2)	$op \longrightarrow_f oq$ w.r.t. $\mathbf{MI}_{\mathcal{H}}$ iff $op \longrightarrow_f oq$ w.r.t. $\mathbf{MS}_{\mathcal{H}}$	(by defn. of $\mathbf{MS}_{\mathcal{H}}$)
(3)	$op \longrightarrow_f oq$ w.r.t. $\mathbf{MS}_{\mathcal{H}} \Rightarrow op \longrightarrow_f oq$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$	(Theorem 5.1)
(4)	$op \longrightarrow_f oq$ w.r.t. $\mathbf{M}_{\mathcal{H}} \Rightarrow op \longrightarrow_f oq$ w.r.t. $\mathbf{MA}_{\mathcal{H}}$	(1,2,3 above) □

COROLLARY 1. Let *p* and *q* be two points in *prog*. If *prog* exhibits $p \longrightarrow_f q$ w.r.t. σ and $\mathbf{M}_{\mathcal{H}}$, then *prog* exhibits $p \longrightarrow_f q$ w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$. □

COROLLARY 2. Let *prog* exhibit $d = p \longrightarrow_f q$ w.r.t. σ and $\mathbf{M}_{\mathcal{H}}$. If *d* is carried by (x, y) w.r.t. σ and $\mathbf{M}_{\mathcal{H}}$, then *d* is carried by (x, y) w.r.t. σ_A and $\mathbf{MA}_{\mathcal{H}}$. □

The notion of embedding also allows dependence to be estimated w.r.t. arbitrarily large sets of stores. Suppose, for example, that a program *P* supports three types of structures: environments, cons cells, and atoms. Suppose, moreover, that *P* supports two identifiers, *x* and *y*. Then a safe estimate of *P*'s dependences w.r.t. to the set of all initial stores can be obtained by evaluating *P* w.r.t. a store with

- one ordinary global environment, *genv*;

- one summary structure, s, of type { **cons**, **atom** } with value $\top^{AT}$;
- two references from *genv* to s of type x and y; and
- two self-references from s to s of type *hd* and *tl*.

This store is shown as Store F in Figure 5.2.

5.3. Using the Determinate Selector Property to Sharpen the Interpretation

In their survey on non-determinacy, Sondergaard and Sestoft distinguish between interpretations that exhibit, and interpretations that fail to exhibit, the ***determinate variable property*** [Son87]. This property is illustrated in the following example program.

$$[1]\ x := 1\ ?\ 2;\ \ [2]\ y := x + x$$

The operator ? is the binary nondeterministic choice operator. In statement [1], for example, "1 ? 2" denotes either 1 or 2, depending on the whims of the implementation. If an interpretation I exhibits the determinate variable property, then this program's interpretation (according to I) assigns either 2 or 4 to y. If I does not exhibit this property, then statement [2] could also assign 3 to y; [2], in effect, is interpreted as "[2] $y := (1\ ?\ 2) + (1\ ?\ 2)$".

Similar concerns arise in the nondeterministic interpretation of pointer languages. Let σ_0, for example, be a store in which x denotes an ordinary location with *two* selectors of type *x.intp*: one to an atom 1, and a second to an atom 2 (*cf.* Figure 5.3, store s_0). The interpretation of "$y := x.intp + x.intp$" would then return three stores. The value of y in these three stores would be 2, 3, and 4, respectively.

This loss of precision can sometimes be avoided by altering the interpretation of selector expressions. Function $selexp_A$, when run on a store σ and an expression *idexp*, currently returns the set of locations that *idexp* denotes in σ. The altered $selexp_A$ changes σ to reflect choices made during the interpretation of *idexp*. Assume, for example, that *idexp* denotes a single structure s w.r.t. a store σ. Assume that s is an ordinary structure that contains two references of type *sel*: one reference r_1 to a structure at location l_1, and a second reference r_2 to a structure at location l_2. The revised $selexp_A$, when run on *idexp.sel* and σ, returns two objects: l_1 *paired* with a store that lacks r_2, and l_2 *paired* with a store that lacks r_1. This pruning of unselected references at ordinary structures is safe, since $\mathbf{M}_{\mathcal{H}}$ is deterministic.

Consider how the revised $\mathbf{MA}_{\mathcal{H}}$ would evaluate "$y := x.intp + x.intp$" w.r.t. σ_0. The evaluation of the first *x.intp* would produce two stores, σ'_0 and σ''_0, in which *x.intp* denoted 1 and 2, respectively. The evaluation of *x.hd* relative to σ'_0 and σ''_0 would determine that *x.hd* must denote 1 w.r.t. s'_0, and 2 w.r.t. s''_0. The interpretation of "$y := x.intp + x.intp$" now yields two stores: one where y is set to 2 and *x.intp* to 1, and one where y is set to 4 and *x.intp* to 2.

A second situation where the revised $selexp_A$ sharpens the interpretation is depicted in Figure 5.3. The version of $\mathbf{MA}_{\mathcal{H}}$ given in Appendix 3 maps store s_0 to stores s_1 and s_2. The adjusted interpretation of $\mathbf{MA}_{\mathcal{H}}$ would return one store in which *x.intp* was 0.

5.4. Related Work

Most of the techniques for analyzing pointer programs described in Chapter 5 are sketched in earlier reports on pointer-program analysis. This dissertation, however, is the first that considers various interpretation and approximation techniques from a unified perspective. A related attempt to unify various tech-

niques for analyzing higher-order functional languages is discussed in [Deu90]. The material presented in Section 5.3 also appears to be new, although a related idea was proposed by Stransky (see below).

The rest of this section discusses related techniques for pointer-program analysis. A discussion of store abstraction strategies is deferred until Chapter 6.

5.4.1. Related abstraction techniques

Each of the abstraction techniques described in Section 5.1.1 has been described in previous papers on program analysis. The particular combination of these techniques described here is new.

The approximate occurrence string is related to the approximate call string of Sharir and Pnueli [Sha81] and to Harrison's (approximate) stack configurations [Har89]. Harrison uses stack configurations to estimate the lifetimes of dynamically allocated objects in Scheme.

The abstract store graph is a direct descendant of the graphs described in [Jon79, Jon81]. Jones and Muchnick, who were concerned with storage-sharing, label summary structures with values that characterize the topology of the replaced region. Other authors that use similar graphs include Pleban [Ple81], Stransky [Str88], Larus [Lar87], and Chase, Wegman and Zadeck [Cha90]. Store graphs are also similar to Chase's *storage containment graph* (SCG) [Cha87] (a descendant of Schwartz's *subpart graph* [Sch75]). The principal differences between store graphs and SCGs are superficial. SCGs, for example, contain two types of edges: one type of edge that denotes a reference, and a second that pairs a structure s with a special node—*i.e.*, a label—that names s's defining point.

Embeddings play a crucial role in other analyses that use graphs to abstract memory. Authors who develop explicit embedding relations include Jones and Muchnick [Jon79, Jon81], Chase [Cha87], Stransky [Str88, Str90], and Chase, Wegman, and Zadeck [Cha90]. The notion of abstraction by embedding is implicit in other work that uses monotone dataflow frameworks to build store graphs: *e.g.*, Ruggieri's work on garbage collection [Rug87, Rug88] and Larus's work on parallelizing Lisp [Lar87].

The distinction between ordinary and summary objects has been drawn by previous authors, notably Jones and Muchnick [Jon79, Jon81], Chase [Cha87], Stransky [Str88], and Chase, Wegman, and Zadeck [Cha90].

Most papers describe analyses that pair every program point with a single abstract store. One exception to this observation is a paper by Deutsch on the analysis of higher-order functional languages [Deu90]. A second is the set-valued interpretation developed in the Jones and Muchnick work on analyzing Lisp-like languages [Jon79, Jon81].

5.4.2. Related interpretations

The immediate precursor of $\mathbf{MA}_{\mathcal{H}}$ is the flow-sensitive semantics for pointer-program analysis developed by Jones and Muchnick [Jon79, Jon81]. This analysis supports a procedure-free subset of Lisp that has a destructive update operator and one type of allocatable object—the cons cell.

One extension of the Jones and Muchnick framework was developed by Pleban [Ple81]. Pleban's analysis supports a subset of Scheme that provides continuations and closures, but does not allow closures to be stored or returned from procedures—thereby avoiding the upward funarg problem.

A second, non-set-valued extension of the Jones and Muchnick framework was developed by Stransky [Str88]. Stransky's analysis supports a subset of Lisp that lacks closures. Stransky obtains a flow-sensitive analysis by using predicates as filters of data: *i.e.*, as assertions that remove paths from stores. Let P, for example, denote the predicate "$x > 0$". Predicate P, when applied to a store s_0, trims paths from s_0 that fail to satisfy the assertion "x may be numeric." Predicate P also trims paths from the stores passed to its true and false control-flow successors, according to whether x is positive or nonpositive along these paths. A path π that paired x with the value -1, for example, would be trimmed from the store passed to P's true consequent. An appreciation of Stransky's technique for interpreting predicates led to the observations about the determinate selector property given in Section 5.3.2.

The technique for interpreting the return statement is similar to the one proposed by Myers [Mye81] and later rediscovered by Jones and Muchnick [Jon82]. This technique, which uses an approximation to the stack to identify a procedure's potential return points, can be contrasted with stack-less techniques that simply assume all possible return paths to be valid (*e.g.*, [Cou78]).

The extended control-flow-graph model of program evaluation is easiest to work with when the example language does not support local variables. In such languages, the return statement does not affect the configuration of a program's memory. Examples of pointer analyses for local-variable-free languages include those by Chase [Cha87], Chase, Wegman, and Zadeck [Cha90], and Larus [Lar87].

Propagating approximate stores through extended control-flow graphs becomes a little more difficult when a language supports nested scopes. The interpretation described in this thesis uses a special set of references to track the local environment. Stransky, who treats the stack and heap as separate objects, maintains a safe estimate of a program's stack by performing comparable folding and unfolding (*pliage et dépliage*) operations on the abstract stack [Str88]. Pleban's evaluator, on the other hand, appears to lack abstract procedure activation records [Ple81]. Pleban suggests that an analysis be run until it duplicates some memory configuration that sits atop the stack.

Cousot and Cousot introduced *estimation operators* (there called widening operators) to ensure the termination of analyses over infinite abstract domains [Cou77]. Another use of widening operators appears in Stransky's thesis [Str88].

5.4.3. Related proofs of correctness

Related proofs of correctness for pointer-program analyses have been given by Jones, Muchnick, Stransky, Deutsch, and Hendren. Jones and Muchnick use Abstract Interpretation to show the safety of two techniques for alias analysis, relative to procedure-free and procedure-supporting dialects of Lisp [Jon79, Jon81, Jon82]. Stransky's thesis sketches, but does not actually give, a proof of correctness for a label-based analysis of a Lisp-like language with dynamic scoping [Str88, Str90]. Deutsch describes an abstract-interpretation-based proof of correctness for a framework for analyzing higher-order functional languages [Deu90]. Hendren's thesis uses a denotational definition of a language as a starting point for demonstrating the correctness of an alias-analysis technique [Hen90].

A second well-established framework for developing program analyses is the monotone dataflow framework of Kildall and Kam-Ullman [Kil73, Kam76]. This second framework predates Abstract Interpretation. The principal reason for using Abstract Interpretation to demonstrate the safety of $\mathbf{MA}_{\mathcal{H}}$ is that show-

ing that $\mathbf{MA}_{\mathcal{H}}$ is a monotone dataflow framework would *not* establish the desired relationship between $\mathbf{MA}_{\mathcal{H}}$ and $\mathbf{MS}_{\mathcal{H}}$; it would merely guarantee that $\mathbf{MA}_{\mathcal{H}}$ has a least fixpoint.

5.4.4. Other graph-based store abstraction techniques

The value $\top^{AT}$ gives a coarse estimate of an atom's value. More refined estimates can be developed from domains of atomic values, intervals, and types. The four-element lattice $(\bot_{AT}, \mathit{nil}, \mathit{non\text{-}nil}, \top^{AT})$ appears in work by Cousot and Hendren [Cou78, Hen90]. More elaborate lattices of approximate values are given by Stransky [Str88, Str90].

A program's evaluation can also be sharpened by extending the distinction between ordinary and summary objects to references. In this extension of the interpretation, it is safe for assignment to replace ordinary references at summary structures, and summary structures at ordinary references. The distinction between ordinary and summary references is drawn by Jones and Muchnick [Jon79, Jon81] and Schwartz (reported in [Cha87]), who replace sets of concrete references with special references labeled *any*. Stransky uses what is tantamount to a typed summary reference to determine when not to remove references from stores [Str88, Str90]. Even sharper characterizations of program evaluation may be obtained by using counts to estimate the number of references that a summary reference represents—or, alternatively, by pairing structures with abstract reference counts [Myc81, Hud87, Str88, Hed88, Cha90].

Summary structures give a coarse estimate of a collapsed section of a store. Sharper estimates can be obtained with annotations that characterize the topology of an abstracted subgraph. Structures have been annotated with values that identify them as abstractions of trees [Myc81], lists (*cf.* Chapter 5 in [Myc81], p. 261 in [Ple81], and Section 4.3 in [Cha90]), and directed acyclic graphs [Jon79, Jon81]. Jones and Muchnick have also used regular tree grammars to capture recurrences generated by programs in functional languages [Jon79, Jon82]. A related idea for using graph grammars to capture regularities in imperative stores is sketched in Chapter 6.

The previous three paragraphs describe more precise abstractions of stores. Most papers on store analysis use simpler abstractions of stores. One simplification of the store graph, the ***alias graph***, is used by Ruggieri to analyze object lifetimes [Rug87, Rug88] and by Larus to compute a program's dependences [Lar87]. Intuitively, a store graph s_0 can be converted into an alias graph by pruning all unshared structures from the frontier of the store (*cf.* Figure 5.9).

Alias graphs can be further compressed by replacing chains of unshared structures with single references. Each replacement reference is labeled with a *path expression* that characterizes the path that it replaces (*cf.* Figure 5.10). The interpretation must then be adjusted to account for edges that are labeled with regular expressions. The adjusted interpretation, in effect, "re-materializes" the elided structures when a program creates new references to what had been unshared structures. This technique is used by Mycroft, and again by Inoue, Seki, and Yagi, to estimate how objects are shared in stores created by applicative programs [Myc81, Ino88]. Hendren uses an equivalent path-compression technique to reduce the size of a comparable representation, the *path matrix* [Hen89, Hen90]. The rows and columns of Hendren's path matrix correspond to a program's identifier expressions; its entries name paths through the heap. Assume, for example, that the ith row in a path matrix M corresponds to the identifier a, and the jth column to the identifier expression *b.next*. Then the $[i, j]$th column in M characterizes the set of paths that link the object referenced by a to the object referenced by *b.next*. Hendren's interpretation is also limited to pro-

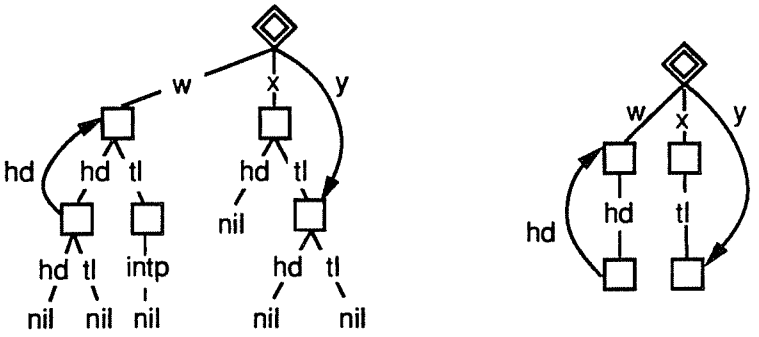


Figure 5.9. A store graph, and its corresponding alias graph.

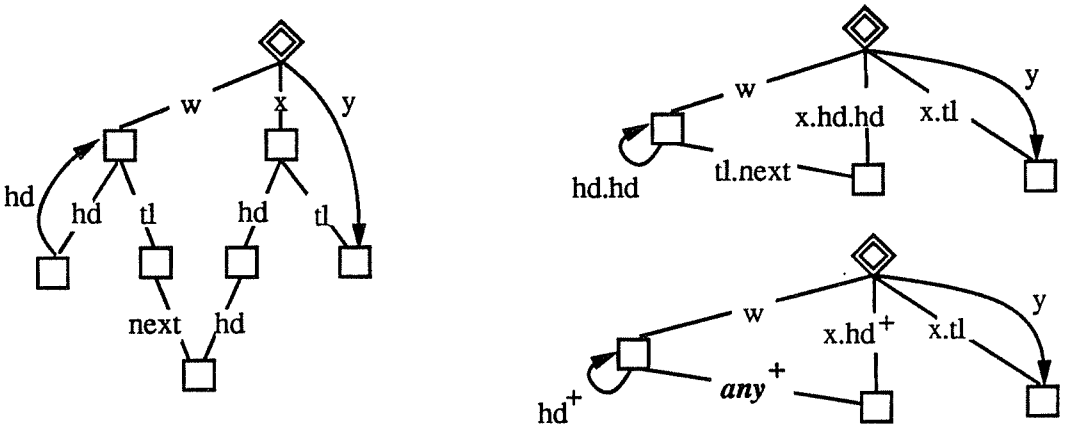


Figure 5.10. Using path expressions to compress alias graphs. The store on the left is an example alias graph. The two stores on the right are compressed alias graphs. The top store is exact; the lower store, approximate.

grams that generate cycle-free stores.

5.4.5. Other graph-based state abstraction techniques

Most authors describe analyses that pair every program point with one or more representations of memory. Chase describes an interpretation that pairs *one* labeled graph with an *entire* program [Cha87]. This graph represents the set of all storage containment graphs (SCG's) that could arise at any point over the course of a program's execution. Chase's analysis forms a program's SCG by merging store graphs produced at separate program points over the course of an analysis. Chase shows that merging SCGs does not lose information about a program's evaluation when the language under consideration supports a copy semantics for assignment statements. An improved version of this idea is developed in [Cha90].

Sagiv, Francez, Rodeh, and Wilhelm describe a logic-based framework for program analysis that augments store graphs with assertions about pointer equalities [Sag90]. These assertions can reduce the amount of space required to record facts about program evaluation. Suppose, for example, that a program manipulates three identifiers named x, y, and z. Suppose, further, that any two, but not all three, of these variables can be aliased. This assertion could be captured in Sagiv's framework by annotating a store graph that showed x, y, and z as aliases with the appropriate assertion about x, y, and z. A related idea is described in a paper by Seo and Simmons, who pair a finite automaton that recognizes a language with a matrix that rules out certain states recognized by the automaton as invalid [Seo88].

5.4.6. Other state abstraction techniques

A 1982 paper by Jones and Muchnick describes a framework for analyzing programs that generate tree-like recursive data structures [Jon82]. This framework uses a representation function and an abstraction of the stack to characterize a program's state. This report also stresses the use of defining-point labels (there called *tokens*) to capture information about a program's behavior.

Various authors partition a program's identifier expressions into sets of equivalence classes [Cou78, Wei80, Gua90, Gua90a]. These algorithms place two identifier expressions in the same equivalence class if they might reference the same structure at a given point in an interpretation.

Coutant describes a technique for tracking a program's aliases in a C-like language [Cou86]. Coutant's analysis monitors the set of memory objects that each of a program's names—arrays, pointers, arrays of pointers, and aggregates—might denote.

Harrison uses sets of closures to model stores [Har89]. Harrison treats a reference x to a record as a binding of a variable x to a new, dynamically allocated function. This function accepts a switch-like argument that either directs it to return the contents of one of its fields, or to update a field and return an updated closure. Selector are then redefined as functions that (1) accept a closure f that represents a structure, and (2) invoke f with the appropriate switch. This treatment of structures makes Harrison's technique, which supports closures, more uniform. Harrison also argues that this approach should allow functions to be returned that characterize the potential dependences of separately compiled procedures.

5.4.7. Other interpretations

Hendren gives an elegant algorithm for estimating a recursive procedure's behavior, w.r.t. a first-order language that lacks mutual recursion [Hen90]. Hendren's algorithm, roughly speaking, pairs a procedure P with a pair of abstract stores $(\sigma_{in}, \sigma_{out})$. This pair represents the assertion that P maps every store that embeds in σ_{in} to a store that embeds in σ_{out}. The following is a sketch of her algorithm:

1. A first estimate of P's behavior, (σ_i, σ_o), is generated by propagating a σ_i passed to P along all recursive-call-free paths through P.
2. An estimate of P's maximal input is computed from σ_i. The initial estimate (σ_i, σ_o) is iterated to obtain a σ'_i that subsumes σ_i.
3. Store σ'_i is then held fixed, and σ_o iterated. If this second iteration produces an estimate of P's maximal output that does not invalidate σ'_i, then the algorithm terminates. Otherwise, the second sequence of iterations caused a store σ_x that did not embed in σ'_i to propagate to a call to P. The algorithm then restarts at step 1 with a new estimate of P's behavior generated from σ_x and σ'_i.

It is clear that Hendren's algorithm can be implemented as a specific iteration strategy over a program's extended control-flow graph. What makes Hendren's formulation of interprocedural analysis appealing is that the strategy is an *explicit* part of her analysis.

Weihl uses information about a reference's type (w.r.t. a strongly-typed example language) to constrain a program's potential aliases [Wei80]. Weihl's algorithm supports programs with procedure-valued variables. An important limitation of Weihl's algorithm is its failure to use information about a program's control flow. Landi and Ryder argue that this produces estimates of a program's aliases that are too coarse to be readily useful [Lan90].

A second algorithm that uses strong typing to constrain a program's potential aliases was given by Ruggieri and Murtagh [Rug87, Rug88]. This algorithm is designed for a strongly-typed, procedure-variable-free language that resembles CLU. The first, intraprocedural pass of the Ruggieri-Murtagh algorithm computes a symbolic estimate of how a program's procedures map inputs to outputs. Rules about a variable's type determine (*e.g.*) whether pairs of inputs could be aliased on procedure entry. The second, interprocedural pass of the algorithm estimates how objects propagate between procedures. Ruggieri and Murtagh argue that their two-pass algorithm represents a reasonable compromise between iterating over a program's extended control-flow graph and analyzing procedures in isolation, using worst-case estimates of inputs.

Other symbolic techniques for pointer-program analysis have been given by Reynolds, Jones and Muchnick, Chase, Larus, and Guarna. Reynolds showed how to develop and solve systems of equations that characterize recursive structures generated by functional programs [Rey68]. Jones and Muchnick later described a similar use of regular tree grammars to estimate the sets of stores generated by a functional program [Jon79, Jon81].

Chase discusses the use of extended store graphs to discover opportunities for speeding fixpoint computations [Cha87]. These graphs, which Chase calls *update graphs*, contain additional nodes that correspond to program points, and additional edges that represent assertions about how pairs of program points share data. Chase uses these graphs to replace a cyclic sequence of assignment statements like

while *pred* **do** $b := a$; $c := b$; $\cdots$ $z := y$; $a := z$ **od**

with a single operation that performs a pessimistic update on the store graph. A related idea appears in Larus's thesis, which discusses the use of ***summary graphs***—graphs that summarize the effect of a set of statements—to speed program analysis [Lar87].

Guarna's technique for program analysis, which uses a semigroup-like algebra of selector expressions to analyze a pointer program's aliases, is discussed in Section 6.6. Other work on pointer program analysis is beyond the scope of this thesis. This includes Harrison's use of object lifetime analysis to estimate a program's dependences, relative to a *call/cc*-free dialect of Scheme [Har89]. This also includes Jouvelot and Gifford's use of type and effect information to reason about program behavior [Jou91], and the use of invariants and meaning-preserving transformations to reason about Lisp-like programs [Jor86, Mas86, Mas90].

OTHER REMARKS ABOUT CHAPTER 5

Various authors (*e.g.*, Jones and Mycroft [Jon86]) credit Sintzoff with the original idea for Abstract Interpretation [Sin72]. Sintzoff argued that a dataflow analysis could be viewed as non-standard interpreta-

tion of a program on an approximate domain. Cousot and Cousot developed the first framework for showing that non-standard interpretations of a programming language were consistent with a language's standard interpretation. Subsequent authors have developed variants of the Cousots' framework, including Mycroft and Nielson [Myc81, Myc83], Nielson [Nie84], Mycroft and Jones [Myc85], and Jones and Mycroft [Jon86]. These papers, which are principally concerned with functional languages, assume denotational definitions of a program's meaning.

Different authors have given different names to the progression of semantics that are used in Abstract Interpretation. The term *instrumented semantics* is due to Neil Jones [*private communication, through Reps*]. Nielson uses the terms *collecting semantics* and *sticky semantics* to refer to a related style of program definition [Nie90]. What the Cousots refer to as a static semantics is referred to by Jones and Mycroft as a *collecting semantics* [Jon86] and by Nielson as a *sticky lifted store semantics* [Nie90]. The term *abstract interpretation* was used by the Cousots to refer to what Nielson calls an *approximation semantics* [Nie84].

6. STRATEGIES FOR ESTIMATING A PROGRAM'S STATES

> *There are, alas, many signs that our field (and most current work in the whole programming languages area as well) is far from scientifically mature. One important problem: there is all too little* research *in the classical meaning of the term, meaning to search systematically through the existing literature for ideas and results [relevant] to one's current goals, even though perhaps expressed in a quite different language or framework. The inevitable result is that many works "reinvent the wheel" and omit highly relevant references to others' work.* —N. Jones [Jon88]

> *If we could first know* where *we are, and* whither *we are tending, we could better judge* what *to do, and* how *to do it.* —A. Lincoln, cited in [Oat77]

The semantics given in Chapter 5, $\mathbf{MA}_{\mathcal{H}}$, uses an estimation function ∇ to ensure that analyses terminate. Chapter 5 assumes that ∇ is an extensive operator that restricts the length of every infinite ascending chain in $pwr(State_A)$. The precise definition of ∇, however, was left unspecified. The primary reason for leaving ∇ unspecified is that the problem of finding a best estimate for arbitrary program's store configurations appears incapable of exact solution. Larus showed that an important subproblem of dependence computation, that of determining a pointer program's aliases, was NP-complete [Lar89]. This result was later amplified by Landi and Ryder, who show that alias computation is NP-complete in languages that support two or more levels of reference indirection [Lan91].

Various heuristics have been proposed for estimating the objects that a pointer program's computation might generate. Sections 6.1, 6.2, 6.3, and 6.4 discuss strategies for estimating labeled stores, sets of labeled stores, occurrence strings and sets of states, respectively. Section 6.5 discusses comments that other authors have made about the potential cost of store approximation. Section 6.6 concludes with a discussion of related work.

Most of the concepts presented in Chapter 6 were originally developed by previous authors. This chapter's primary contributions are this survey of store approximation techniques presented in Section 6.1 and the variant of the *k*-limiting technique for store approximation described in Section 6.1.1.

6.1. Abstracting Labeled Stores

The number of structures that a (labeled) store may contain must be bounded if a program is to have a terminating interpretation w.r.t. $\mathbf{MA}_{\mathcal{H}}$.[5] More precisely, the stateset estimation operator ∇ must restrict the number of structures that a program's *while* loops and recursive procedures add to a store. To ensure a safe result, ∇ must also map every store σ that grows too large to a bounded store that subsumes σ.

Various techniques have been proposed for limiting the size of a store graph σ. In this thesis, such techniques will be treated as special instances of the following three-step algorithm for store reduction:

1. Every inaccessible structure is removed from σ.
2. A ***partitioning strategy*** is used to divide the updated σ into $n+1$ sets of structures. One set of structures is left unchanged by the algorithm. The other n sets of structures are replaced by representative

[5] The number of references in a store σ must also be bounded, but this can be accomplished by first limiting the number of structures in σ, and then stipulating that there can be no more than one reference of a given type between any two structures in σ.

structures. Refer to these sets as *protected* and $unprotected_1 \cdots unprotected_n$, respectively.

3. A ***reduction strategy*** is then used to limit the size of each $unprotected_i$. Each subgraph of σ induced by an $unprotected_i$, G_i, is replaced with a bounded graph that subsumes G_i. The use of a replacement graph that subsumes G_i ensures that the updated σ subsumes the original store.

This algorithm, which will be referred to as the ***divide-and-shrink*** algorithm for limiting store size, is depicted in Figure 6.1. This algorithm returns a bounded approximation to a store σ when its partitioning strategy limits the value of n and the size of *protected*, and its reduction strategy limits the size of each replacement graph. Specific implementations of divide-and-shrink differ according to the strategies used to partition and reduce stores. Accordingly, the rest of this section discusses the merits and weaknesses of various store partitioning and reduction techniques.

6.1.1. Partitioning strategies

The scheme for classifying partitioning strategies used here divides these schemes into four categories:

- Strategies that use labels.
- Strategies that use labels, and also exploit information about a store's paths.
- Strategies that use paths.
- Strategies that use paths, and also exploit information about a store's labels.

Since stores are essentially collections of paths and labels, this classification scheme may appear to do little more than state the obvious. The author, nevertheless, has found this taxonomy useful for thinking about the various approaches to store abstraction.

Label-driven partitioning strategies place two structures s_1 and s_2 in a common partition if s_1 and s_2 have related labels. The basic label-driven partitioning strategy, which was first described by Hudak [Hud86, Hud87] and Chase [Cha87], uses a store σ's allocation-point labels to partition σ. Let a program P, for example, contain n statements that allocate structures. Assume, furthermore, that the abstract interpretation of P labels every structure s with the name of that point that allocates s. Let σ be a store generated during a computation that involves P. The basic label-driven partitioning scheme partitions σ into at most n sets of structures, and places all structures allocated at a common program point a common set. In the basic version of label-driven partitioning, there is no special, "protected" set of structures; every partition is a potential candidate for abstraction.

The basic label-driven partitioning strategy has two pleasant properties. This strategy places objects allocated at different statements into different partitions. This allows the second phase of divide-and-shrink to create approximate stores whose summary structures are each labeled with exactly one creation point. This is potentially advantageous, since merging structures with different allocation points loses information about an abstract structure's allocation site. This partitioning strategy is also monotonic. Assume, for example, that σ and σ' are arbitrary stores such that $\sigma \sqsubseteq \sigma'$ by an embedding f. If the label-driven technique puts s_1 and s_2 in a common (unprotected) partition, then it must also put $f(s_1)$ and $f(s_2)$ in a common partition. This property of label-based partitioning allows the development of a monotone store abstraction operator. This observation about the monotonicity of label-driven partitioning also holds for the other label-driven strategies described below.

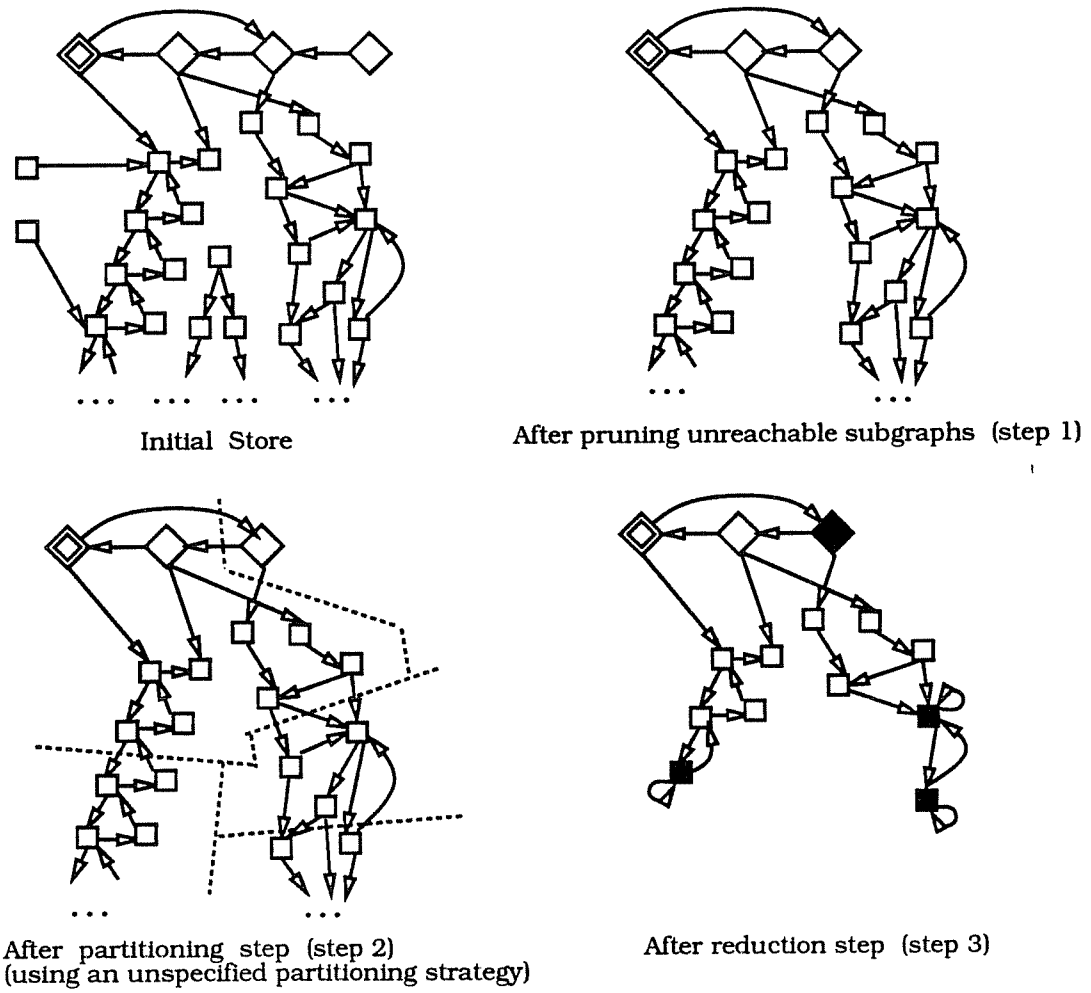


Figure 6.1. The three-step divide-and-shrink algorithm for limiting store size. The first step in the divide-and-shrink algorithm removes structures that are unreachable from the global environment. The second step identifies one protected and n unprotected sets of structures in σ (here, the protected region contains the global environment). The third step replaces the subgraph G induced by every unprotected set of structures with a bounded, representative graph that subsumes G.

The basic label-driven partitioning technique has two limitations. An "ideal" partitioning strategy splits every store into sets of "algorithmically equivalent" structures: structures that are treated in roughly the same way over the course of a computation. The basic label-driven strategy assumes, in effect, that all structures allocated at a given program point are treated in a uniform manner. Real programs, however, do not necessarily abide by this assumption. Consider, for example, the effect of using the basic label-driven strategy to analyze a program that has exactly one allocation site:

procedure *allocate*(*ptr–to–struct*, *type*) ; *ptr–to–struct* := **new**(*type*) **end**

It would almost certainly be better, in such an extreme, to use *any* other partitioning strategy. A second

limitation of this strategy is the technique's tendency to lose information about a store's topology. There is no reason, in other words, to suppose this strategy will place adjacent or proximate structures—even structures in the same list—in a common partition.

Various extensions of the basic label-driven partitioning technique have been defined. Most give finer characterizations of a program's behavior. One, described by Hudak, uses fragments of occurrence strings to partition the store [Hud86, Hud87]. Assume, for example, that an interpretation labels every structure s allocated in a procedure P with *two* program points: the statement at which s was allocated, and the statement that invoked the particular call to P that allocated s. Hudak's (second-order) partitioning strategy would then group two structures iff they had the same two-component label. Hudak's technique would probably prove useful for programs (like the one described in the previous paragraph) that use a few server-like procedures to allocate structures.

Other authors obtain finer characterizations of list-like structures by labeling the kth structure allocated at a program point p with the value $[p, k]$. A store's structures are first partitioned by allocation site. The set of points allocated at a given site p is then partitioned by counter. Stransky, for example, protects the first k structures allocated at each program point [Str88]. This allows the divide-and-shrink algorithm to construct abstract stores that give an exact characterization of portions of certain lists (*e.g.*, the heads of lists that are built top-down from structures allocated at a single statement.) Another counter-based partitioning technique, developed independently by Bodin [Bod90] and Gohkale and Smith [Goh90], places structures that have the same program point and counter mod k into a common (unprotected) partition. This scheme has been used to estimate the distance of a loop-carried data dependence.

A third extension of label-driven partitioning supports set-valued labels. (*N.B.*: set-valued labels arise during the computation of input, anti-, and def-order dependences.) This technique, which is related to partitioning techniques described by Larus and Hilfinger (see below), places two structures s_1 and s_2 in the same partition iff either

- the labels of s_1 and s_2 have a non-empty intersection, or
- there exists a third structure s_3 such that s_1 and s_3 are placed in the same partition, and the labels of s_3 and s_2 have a non-empty intersection.

Consider, for example, a set of structures s_1, s_2, s_3, s_4, and s_5 whose reading-point labels are { [1],[2] }, { [2],[3] }, { [3],[4] }, { [5] }, and { [5],[6] }, respectively. The partitioning strategy just described places s_1, s_2, and s_3 into one partition, and s_4 and s_5 into a second.

Still other variants of the basic label-driven partitioning strategy can be defined that use some combination of label, type, and value information to partition the store.[6]

A second type of ***label-driven partitioning strategy*** was developed by Larus and Hillfinger [Lar88, Lar89]. This technique assigns every newly allocated structure s a label that characterizes the ***path*** along which s was added to the store. Assume, for example, that a program is about to build a list of cons

[6] Strictly speaking, it might be more proper to talk about *attribute-driven* (rather than *label-driven*) partitioning. The word "label" has been chosen for historical reasons.

cells that is to be referenced by a variable x. Assume, furthermore, that this list is being built in a top-down fashion. Then the first cons cell added to this list would be labeled *x.hd*. Successive cells would be labeled *x.hd.hd*, *x.hd.hd.hd*, *etc*. A list at x of indeterminate length would be terminated with a summary structure labeled $x.hd^{k-1}.hd^*$, where k is an arbitrary limit on the number of selectors in a label.

The first step in the Larus-Hilfinger abstraction technique replaces every label that contains k or more selectors with a related k-selector-long regular expression. This new regular expression is an abstract label that subsumes the original label. Assume, for example, that k is 3. Then Larus's label-replacement algorithm would replace *x.hd.hd.hd.hd.tl* with $x.hd.hd^*.tl$, and *tmp.a.b.c.* $\cdots$ *x.y.z* with *tmp.a.*$(b \mid c \cdots x \mid y)$*.z*. The partitioning step then places two structures s_1 and s_2 in a common partition if their labels subsume a common selector expression, or if there exists a third structure s_3 such that s_1 and s_3 are placed in the same partition, and the labels of s_3 and s_2 subsume a common selector expression.

The Larus-Hilfinger partitioning strategy is monotone. A second pleasant property this strategy is its tendency to preserve a store's topology: *i.e.*, to place structures that are adjacent or proximate in a common partition.

Two important limitations of the Larus-Hilfinger partitioning technique are shared by the basic path-driven partitioning technique described below. The first is the technique's failure to group structures that are operated on (*e.g.*, allocated) by a common program point. The second is the k-limiting assumption, which poses two serious problems for implementations of this technique:

- How can the need for a more precise analysis be balanced against the potentially exponential increase in the size of the store that results from an increase in k?
- How can an appropriate value be chosen for k?

Possible solutions for both problems are proposed below, in the discussion of path-driven partitioning.

A third limitation of this technique is its strategy for labeling structures that are prepended to existing structures. The original version the Larus-Hilfinger algorithm labeled such structures with values that characterize the new structure's referents [Lar88]. For example, the two structures created by the evaluation of **cons**(x, **cons**(x, y)) are labeled $<x, y>$ and $<x, <x, y>>$, respectively. This approach apparently proved unsatisfactory in practice, since it was later replaced with a second labeling strategy. This second strategy, given in Larus's thesis, assigns to each prepended s a hybrid label that names the program point at which s is created [Lar89]. The formula for creating this label, which is somewhat complicated, is also given in Larus's thesis. This second labeling technique results in a compromise partitioning strategy that groups some structures by proximity, and others by program point.

Path-driven partitioning strategies use a store's topology to restrict its growth. The basic path-driven partitioning strategy, ***k-limiting***, is discussed in Tenenbaum's thesis on type determination [Ten74] and in the Jones-Muchnick reports on analyzing Lisp-like languages [Jon79, Jon81]. The Jones and Muchnick version of k-limiting, roughly speaking, first partitions a store σ into two sets of structures:

1. The set of all ordinary structures in σ that can be reached from σ's global environment along a summary-structure-free path that contains k or fewer references.
2. The set of all other structures in σ.

Structures in set 1 are placed in the set of protected structures, *protected*. Structures in set 2 are first placed in a single set of unprotected structures, *unprotected*. Set *unprotected* is then partitioned into maximal connected components. In particular, let s_1 and s_2 be two structures in *unprotected*. Then s_1 and s_2 are placed in the same partition iff either

- Structures s_1 and s_2 are *connected*: that is, if s_1 references s_2, or vice-versa; or
- there exists a structure s_3 such that s_3 is in *unprotected*, s_1 and s_3 are placed in the same partition, and s_3 and s_2 are connected.

The number of sets generated by this partitioning of *unprotected* into maximal, connected subgraphs can be bounded by restricting the *fanout* of a store's ordinary structures—the number of references of a given type that may be situated at a given structure. Jones and Muchnick, for example, generate abstract stores that have no more than one reference of a given type at any ordinary structure—and no more than one reference from any summary structure to any other structure in the store.

Path-driven partitioning strategies have one important advantage over label-driven partitioning strategies: they allow the creation of abstract stores that preserve an arbitrarily large, contiguous region of a store unchanged. The basic path-driven partitioning strategy also has three important problems:

1. The size of a k-limited abstract store is potentially exponential in k.
2. No good rationale has yet been given for choosing a specific *cutoff*—*i.e.*, an appropriate value of k—for a given analysis.
3. Structures that are operated on (*e.g.*, allocated) by comparable program points might not be grouped together.

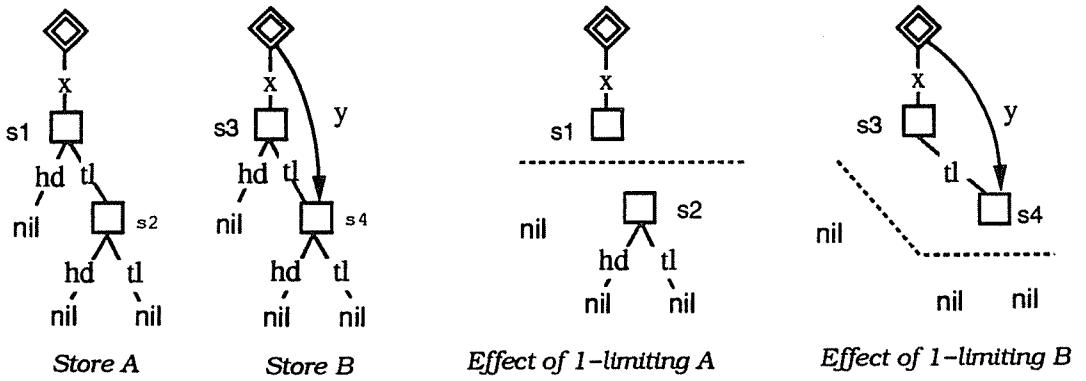


Figure 6.2. The path-based partitioning operator is not monotonic w.r.t. $\sqsubseteq$. A $\sqsubseteq$ B by an embedding that maps s_2 to s_4. The two figures on the right depict the effect of 1-limiting A and B, respectively. Structures above the dotted lines are placed in the protected partition; structures below the dotted line are placed in unprotected partitions. Note that s_2 is placed in an unprotected partition, and s_4 in the protected partition.

A fourth limitation of path-driven partitioning operators is that they are not monotone w.r.t. $\sqsubseteq$ (*cf.* Figure 6.2). This last limitation, however, does not interfere with the use of a path-based partitioning strategy in any essential way.[7]

The concern about the exponential blowup in k can be addressed by imposing *asymmetric* limits on a store's extent. It seems reasonable to use *any* finite, *prefix-closed* set of identifier expressions to partition a store into sets of protected and unprotected structures:

DEFINITION. A set of non-empty identifier expressions *expset* is ***prefix-closed*** iff *expset* contains ε, the empty identifier expression, and *expset* contains *idexp* whenever it contains *idexp.sel*. □

Assume, for example, that an analysis seeks to determine how a program operates on structures referenced by y, and the first k structures in lists referenced by z. The set of identifier expressions *pathset* = { ε, y, z, $z.hd$, $z.tl$, $\cdots$, $z.hd^k.hd$, $z.hd^k.tl$ } could be used to obtain careful estimates of operations on these structures—and weaker estimates of operations on other structures. More precisely, the partitioning step of divide-and-shrink would protect every structure named by *pathset*—and leave all remaining structures unprotected.

The concern about how to generate appropriate cutoffs—*e.g.*, useful sets of prefix-closed expressions—can be addressed by *using stores generated during an analysis to guide the partitioning.* More specifically, the stores that first propagate to a recursive construct r could be used, in conjunction with the form of r, to bound the states generated by r. Suppose, for example, that a store σ reaches a while loop L during the course of an analysis. Suppose, for simplicity, that L contains no other loops, no call statements, and exactly one allocation site. To determine a suitable cutoff for σ, one might start with a prefixed-closed *pathset* that names every structure in σ. A set of prefix-closed identifier expressions for evaluating L w.r.t. σ could then be determined by evaluating L once; determining the shortest identifier expression that names the newly allocated structure in the updated σ; adding this identifier expression to *pathset*; prefix-closing the updated pathset; and then using the updated *pathset* to analyze L w.r.t. σ.

The concern about the failure of ***path-driven partitioning*** to group program points ***with related labels*** can be addressed by partitioning the store yet another time. This third partitioning of the store would group together all structures in a given maximal component that have related labels.

These proposed adjustments to the basic k-limiting strategy are an original contribution of this dissertation. It is important to point out that this adjusted k-limiting technique has neither been implemented, nor tested. The principal reason for discussing it here is that a recent paper on store reduction by Chase, Wegman, and Zadeck argues that the basic k-limiting technique is not practical, for the three reasons given above [Cha90]. It is hoped that these arguments on behalf of a modified k-limiting technique show that path-driven partitioning strategies may prove viable for store reduction.

---

[7] Path-driven partitioning can be shown to be monotone w.r.t. a more restrictive subsumption relation that is one-to-one over part of its range. The use of such a relation, however, restricts an analysis's ability to collapse sets of stores into single, representative stores.

6.1.2. Reduction strategies

Let σ be a store, $unprot = \{ s_1, \cdots, s_n, \cdots \}$ a set of structures in σ, and G_{unprot} the subgraph of σ induced by *unprot*. A *reduction strategy* is a rule that transforms σ into a related graph in which the s_i's are replaced with a bounded number of structures. Intuitively, σ is reduced by first replacing G_{unprot} with a representative G_{new}, and then replacing references to structures in G_{unprot} with comparable references to structures in G_{new}.

Reduction strategies are classified in this thesis according to the form of the G_{new} that replaces G_{unprot}. Two types of strategies are discussed below: those that generate *pointwise* representations of pruned subgraphs, and those that generate *structured* representations of pruned subgraphs.

The basic ***pointwise reduction strategy***, which will be referred to as ***labeled store condensation***, replaces G_{unprot} with a G_{new} that contains one structure. Intutively, G_{new} is created by condensing all structures in G_{unprot} into a single, representative structure s_{new}, and then replacing all paths in σ through *unprot* with comparable paths through s_{new}. More formally, let

* $kind(s_i)$ denote the kind of a structure s_i (*i.e.*, **ordinary**, **summary**);
* $type(s_i)$ denote the type of s_i;
* $label(s_i)$ denote the label of s_i;
* $value(s_i)$ denote the atomic value of s_i (or $\perp_{AT}$, if **atom** $\notin type(s_i)$; and
* (s_i, sel, s_j, l) denote a reference from structure s_i to structure s_j of type *sel* with label *l*.

Then $kind(s_{new}) =$ **ordinary**, iff $n = 1$ and s_i is an ordinary structure, and **summary** otherwise;
$type(s_{new}) =$ **union_from** $s_i \in unprot$: $type(s_i)$;
$label(s_{new}) =$ **union_from** $s_i \in unprot$: $label(s_i)$; and
$value(s_{new}) =$ **join_from** $s_i \in unprot$: $value(s_i)$

where a **join_from** $b = a$ iff $a = b$ or $b = \perp_{AT}$, and $\top^{AT}$ otherwise. The references in the updated σ are computed as follows:

1. Every reference (s_i, sel, s_j, l), where $s_i \in unprot$, is replaced with the reference (s_{new}, sel, s_j, l).
2. Every reference (s_i, sel, s_j, l), where $s_j \in unprot$, is replaced with the reference (s_i, sel, s_{new}, l).
3. The updated σ created by steps 1 and 2 may have references that have the same type and the same endpoints. (This happens, for example, when σ contains two references (s_x, sel, s_i, l_i) and (s_x, sel, s_j, l_j), where s_i and s_j are in *unprot*.) Step 3 replaces every set of references of the form $\{ (s_i, sel, s_j, l_1), \cdots (s_i, sel, s_j, l_k) \}$ with the reference $(s_i, sel, s_j, l_1 \cup \cdots \cup l_k)$.

Strategies for estimating occurrence strings (*cf.* §6.3) can then be used to limit the size of the resulting labeled store's labels.

Let *ls* denote a labeled store, and *unprot* an arbitrary set of structures in *ls*. Let $condense(ls, unprot)$ denote the labeled store obtained by condensing *ls* w.r.t. *unprot*. The assertion that $ls \sqsubseteq condense(ls, unprot)$ follows directly from the definitions of *condense* and $\sqsubseteq$. The assertion that *condense* is confluent—roughly speaking, that $condense(condense(ls, \{ s_1 \}), \{ s_2 \}) \sim condense(condense(ls, \{ s_2 \}), \{ s_1 \})$—can also be demonstrated when the label-estimation strategy is confluent. (Confluence proofs for similar condensation operators are given by Chase [Cha87] and Stransky [Str88].)

Variants of *condense* appear in work by Schwartz [Sch75], Jones and Muchnick [Jon79], Pleban [Ple81], and Stransky [Str88]. Schwartz and Jones and Muchnick use a *condense* operator that replaces *all* references from s_{new} to a given structure s with a single reference of type *any*. Jones and Muchnick, Pleban, and Stransky use a *condense* operator that *removes* all references from s_{new} to s_{new} from the condensed graph. (Later work by Stransky [Str90] uses a *condense* that resembles the one described here.)

The *condense* operator has an important limitation: a naive pointwise condensation operator discards useful information about a store's structure. This point is illustrated in Figure 6.3, which depicts an n-element list (left-hand store) and the store produced by condensing all cons cells in this list (right-hand store). The condensed store, which subsumes the list, can be used in place of the list to obtain a safe estimate a program's behavior. This condensed store, however, does not produce sharp estimates of a program's behavior; it cannot, for example, be used to infer that the pair of selector expressions such as $x.hd^k.tl$ and $x.hd^j.tl$ access different elements of the original list when $j \neq k$.

One technique for avoiding this loss of information, discussed in Chapter 5, annotates every s_{new} that replaces an acyclic G_{unprot} with a label that identifies the topology of G_{unprot}: *e.g.*, identifies s_{new} as a condensed tree, dag, or list. A related idea, developed by Hendren, restricts the topologies of the subgraphs that a summary node can represent [Hen89, Hen90]. Hendren first assumes that every store that a given program can generate is a directed acyclic graph (*dag*) that contains a bounded number of shared struc-

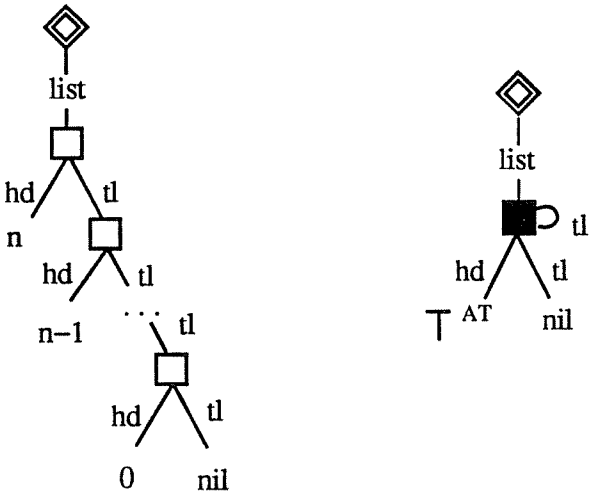
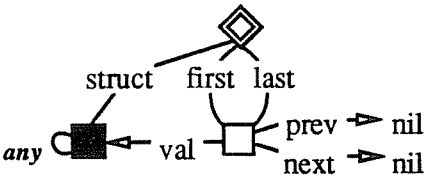
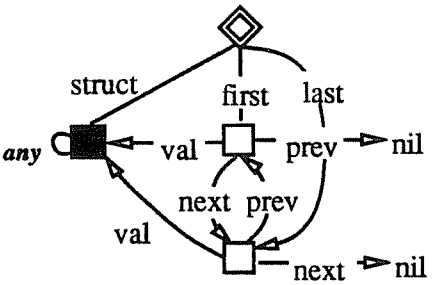


Figure 6.3. Pointwise reduction strategies lose information about a program's structure. The store on the right is obtained by condensing the set of all cons cells in the left-hand store into a single structure. The abstract store cannot be used to infer (*e.g.*) that *x.hd* and *x.tl.hd* represent distinct structures in the left-hand store.

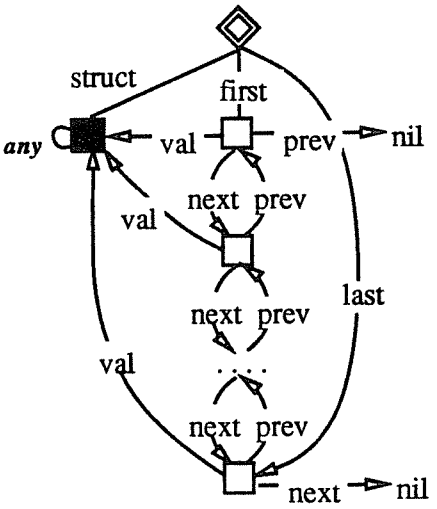
```
        struct listelt  is <prev,value,next>;
        first := new(listelt) ;
        first.prev  :=  nil ;    first.value := struct ;    first.next := nil ;
        last := first ;
[L]     while  pred  do
            last.next :=  new(listelt);
            last.next.prev := last;    last.next.val := struct;    last.next.next := nil ;
            last := last.next
        od
```



Store reaching first iteration of L



Store reaching second iteration of L



Store reaching nth iteration of L

Figure 6.4. An example program, together with the set of doubly-linked lists that it generates. (The example is treated as a procedure-free program, for simplicity.) *struct* is a region of unknown structure in the initial store.

tures.[8] This assumption allows Hendren to partition a store into a contiguous *dag* that contains the global environment, and a bounded number of arbitrarily large trees that are rooted at the perimeter of this *dag*. Hendren's version of divide-and-shrink leaves the *dag* intact, and prunes every tree on the perimeter of the *dag*. This allows Hendren to assume that every reference that points to a structure that lies outside this *dag* references a tree-like summary structure—and to give an interpretation that "materializes" and "unfolds" these structures at need.

The qualified-summary-structure approach to store reduction has the following important limitation: it forces an analysis to anticipate the types of regular structures that a computation might generate. This is

[8] Hendren treats the stack and the heap as distinct objects. A comparable assumption for the representation used here is that stores are dags, up to the reference that identifies a store's current local environment. Her analysis returns failure when it detects that a program might not satisfy this restriction. It must be also emphasized that this account of Hendren's algorithm represents this author's translation of her path-matrix-based analysis into the store-graph framework. Even so, it is believed that this discussion accurately captures the spirit of her work.

unfortunate, since programs can generate many sorts of regular structures. The program given in Figure 6.4, for example, builds a doubly-linked list of references to an unknown set of structures, and maintains an auxiliary reference to this list's final element. A more general technique for abstracting regular data structures that can capture such recurrences is illustrated in Figure 6.5. This technique augments abstract store graphs with ***graph rewriting rules***: rules that characterize families of graphs [Nag79]. Each of the rules shown in Figure 6.5 has the form "*X* rewrites to *Y*", where *X* is a fragment of a graph that contains one structure, *s*, and a set of edges that are incident on *s*.

It is not hard to extend $\mathbf{MA}_{\mathcal{H}}$ to support this style of graph-rewriting rule; the family of graphs that such rules generate can be enumerated in a straightforward manner (*cf.* [Hab86]). A much more challenging problem is the development of ***structured reduction strategies***—pattern-recognizing *condense* operators that can (*e.g.*) automatically generate the productions depicted in Figure 6.5. This observation is supported by the (unpublished) experience of Larus and Wegman, who have investigated the use of graph grammars in program analysis [*private communication*]. The author is currently investigating the task of developing structured reduction strategies. One promising approach involves the use of incremental characterizations of program evaluation to develop the requisite productions.

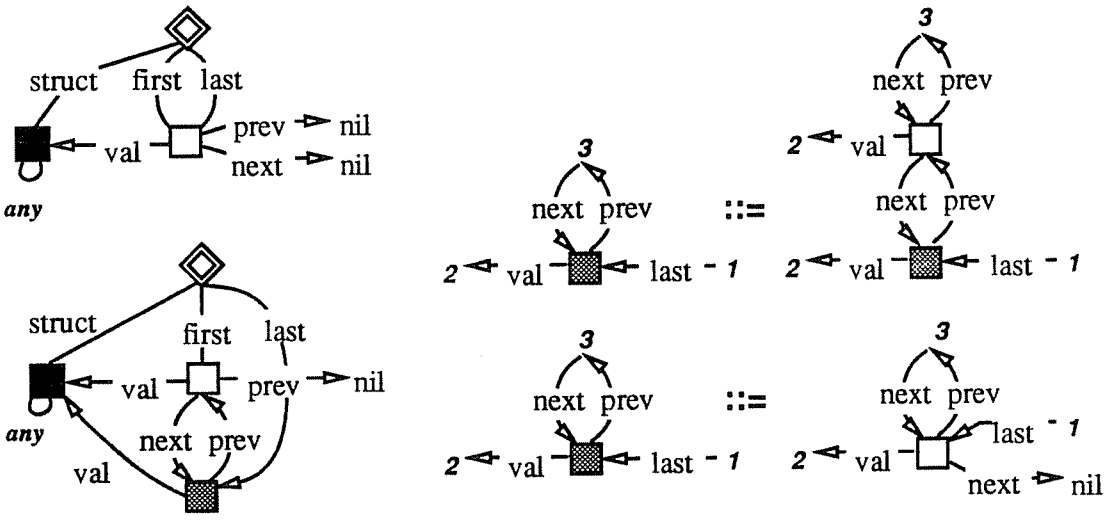


Figure 6.5. A concise, sharp representation of the stores that reach loop *L* in the example program in Figure 6.4. The two productions are *graph rewriting rules*—rules for generating new stores through the replacement of nonterminals (shaded structures) with subgraphs. Boldface numerals relate the structures that anchor the references on the left-hand sides of productions to the structures that anchor the references on the right. (In this figure, **1** denotes the global environment; **2**, the summary structure; and **3**, structures in the doubly-linked list.)

6.2. Abstracting Sets of Labeled Stores

If ∇ bounds the number of labeled stores and occurrence strings that an analysis can generate, then the number of states that reach a given program point must also be bounded—and $\mathbf{MA}_{\mathcal{H}}$ will terminate. There are, however, practical reasons for further limiting the number of labeled stores that reach a given program point. Considerable space may be needed to store the set of all store graphs that an analysis generates; limiting the number of store graphs that can reach a given program point may considerably reduce an analysis's use of storage. Restricting the number of stores that reach certain program points should also speed the average analysis—if one assumes that an analysis's running time is typically proportional to the number of stores that it manipulates.

One technique for reducing the size of set of labeled stores removes redundant labeled stores from this set. Assume, for example, that set *lsset* contains a labeled store, ls_k, that subsumes a second $ls_j \in lsset$. Then *lsset* can be replaced by $lsset - \{ ls_j \}$ with no loss of information. The principal limitation of this technique is that it cannot be used to reduce an *lsset* that does not contain redundant labeled stores.

A second technique for reducing the size of an *lsset* uses a ***graph merging*** operator to replace a pair of labeled stores with a third labeled store that subsumes both. The basic graph-merging operator, *merge*, is related to that variant of divide-and-shrink algorithm that uses the pointwise *condense* operator to reduce store size. The four-step *merge* operation is defined as follows:

1. Let *ls* and *ls'* be a pair of labeled stores, and *struct* and *struct'* the structures in *ls* and *ls'*, respectively. The merging of *ls* and *ls'* first partitions *struct* $\cup$ *struct'* into sets of *comparable* structures. (Criteria for partitioning *struct* $\cup$ *struct'* are discussed below.) Let $related_1 \cdots related_k$ be the set of structures obtained from this partitioning.
2. The second step of *merge* computes the structures in *ls''*, the merged version of *ls* and *ls'*. Each structure s_i placed in *ls''* represents all structures in one of the $related_i$. More formally, the s_i that represents a given $related_i$ satisfies the following equations:

   $kind(s_{new}) =$ **summary**, if either $related_i$ contains summary structures, or more than one ordinary structure from *ls*, or more than one ordinary structure from *ls'*; otherwise, **ordinary**
   $type(s_{new}) =$ **union_from** $s_i \in related_i : type(s_i)$
   $label(s_{new}) =$ **union_from** $s_i \in related_i : label(s_i)$
   $value(s_{new}) =$ **join_from** $s_i \in related_i : value(s_i)$

3. The third step of *merge* places references in *ls''*. Each reference added to *ls''* represents a reference in one of the original graphs. A reference (s''_i, sel, s''_j, l) is added to *ls''* for every reference (s_a, sel, s_b, l) in *ls* (and *ls'*) such that s_a is replaced by s''_i and s_b by s''_j.
4. The final step of *merge* replaces every set of references in *ls''* of the form $\{ (s''_i, sel, s''_j, l_1), \cdots (s''_i, sel, s''_j, l_k) \}$ with the reference $(s''_i, sel, s''_j, l_1 \cup \cdots \cup l_k)$.

The claim that *ls''* subsumes both *ls* and *ls'* follows from the definitions of *merge* and subsumption.

Various strategies can be used to partition structures in two merged stores. These strategies can once again be classified according to whether they use labels, paths, or some combination of the two to group structures.

The basic *label-driven* strategy places two structures s_1 and s_2 in the same partition iff they have the same allocation labels. Stransky discusses the allocation-label-driven graph merging in considerable detail [Str88, Str90]. The extension of this partitioning strategy for set-valued labels groups s_1 with s_2 iff $label(s_1) \cap label(s_2)$ is non-empty, or there exists an s_3 such that s_1 is grouped with s_3 and $label(s_3) \cap label(s_2)$ is non-empty.

Larus's *labels-plus-paths* technique groups s_1 with s_2 if s_1's and s_2's labels subsume a common identifier expression, or if there exists an s_3 such that s_1 is grouped with s_3 and s_3's and s_2's labels subsume a common identifier expression [Lar89].

Let *pathset* be a set of prefix-closed identifier expressions. The basic *path-driven* partitioning strategy groups s_1 with s_2 iff

- neither s_1 nor s_2 is named by an identifier expression in *pathset*; or
- both s_1 and s_2 are named by identifier expressions in *pathset*, and either
  - s_1 and s_2 are named by the same *idexp* $\in$ *pathset*, or
  - there exists an s_3 such that s_1 is grouped with s_3, and s_3 and s_2 are named by the same *idexp*.

Straightforward refinements of this technique repartition (some or all of) the $related_i$ into sets of structures with related labels, values, or types. Figure 6.6 illustrates the use of a *path-plus-labels* partitioning strategy to create a store that subsumes two related stores. Structures in s_1 and s_2 have been partitioned w.r.t. { ε, *w*, *x*, *y*, *z*, *z.hd*, *z.tl*, *z.tl.hd*, *z.tl.tl* }, and then repartitioned into sets of objects that have the same type.

The earlier discussion about the merits and limitations of store-partitioning strategies (*cf.* §6.1.1) applies to the four strategies just described.

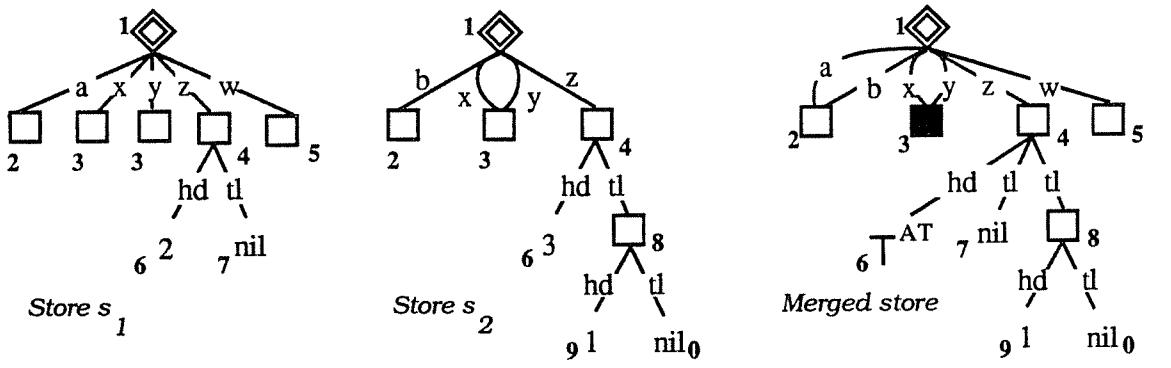


Figure 6.6. Using a path-plus-labels store-merging operator to compute a store that subsumes two stores, s_1 and s_2. $s_1 \cup s_2$ was partitioned w.r.t. { ε, *w*, *x*, *y*, *z*, *z.hd*, *z.tl*, *z.tl.hd*, *z.tl.tl* }. Small, boldface numbers show how the strategy groups structures in $s_1 \cup s_2$, and replaces them with new structures in the merged store.

6.3. Abstracting Occurrence Strings

Language $\mathcal{H}$ lacks procedure-valued variables. A straightforward technique for estimating occurrence strings (*i.e.*, augmented stack configurations) can therefore be derived from a program's intraprocedural and interprocedural control dependences. The technique developed here reduces every occurrence string to a canonical regular expression. These regular expressions, *very* roughly speaking, are strings of the form $(a_1 \mid a_2 \mid \cdots \mid a_\alpha)^+ (b_1 \mid b_2 \mid \cdots \mid b_\beta)^+ \cdots (z_1 \mid z_2 \mid \cdots \mid z_\omega)^+$ where each set of parenthesized program points represents a maximal set of mutually recursive procedure *calls*, and (*e.g.*) at least one of the b's is control-dependent on one of the a's—but not vice-versa. A more precise characterization of this canonical form uses the notion of an *approximate-occurrence-string tree*:

DEFINITION (***unfolding sites***). Let P be a program. Program P's *unfolding sites* are the set of program points in P that represent either call statements, or predicates of loops. □

DEFINITION (***unfolding-site graph***). Let P be a program and us_P be P's unfolding sites. An *unfolding-site graph* for P is a graph G_P that depicts control dependences among P's unfolding sites. Specifically, G_P contains one node labeled p for every p in us_P. Graph G_P also contains one edge $p \longrightarrow q$ for every pair of program points p and q in us_P such that $r_1 \longrightarrow_c \cdots \longrightarrow_c r_k$, where $k \geq 2$, $r_1 = p$, $r_k = q$, each $r_i \longrightarrow_c r_{i+1}$ corresponds to *either* an intraprocedural *or* an interprocedural control dependence (*cf.* §3.2.1), and $r_i \notin us_P$ for all i between 2 and $k-1$, inclusive. □

DEFINITION (***approximate-occurrence-string tree***). Let P be a program, and G_P be P's unfolding-site graph. An *approximate-occurrence-string tree* for P is obtained from G_P by replacing every maximal strongly connected component of G_P with a single, representative, self-edge-free node. Let c, for example, be a maximal strongly connected component of G_P. If c contains one node labeled p, then the node that replaces c is labeled p^+. If c contains n nodes labeled $p_1 \cdots p_n$, then the node that replaces c is labeled $(p_1 \mid \cdots \mid p_n)^+$. □

Figure 6.7 depicts an example program, together with its unfolding-site graph and approximate-occurrence-string tree. A program P's unfolding sites are exactly those program points that can appear in P's occurrence strings. A program P's unfolding-site graph is a variant of its ***procedure call graph*** (*cf.* § 10.8, [Aho86]) that depicts unfolding sites rather than procedures. Every path through P's unfolding-site graph that begins at $\mathbf{initial}_2$, the site of the initial call to procedure *main*(), corresponds to an occurrence string that might arise during an execution of P.

Every path through P's approximate-occurrence-string tree that begins at $\mathbf{initial}_2$ estimates a set of occurrence strings that might arise during an execution of P. More importantly, every occurrence string $o_1 \cdots o_n$ that might arise during an execution of P can be reduced to a path through this tree. Specifically, let $f(o_k)$ denote that node in P's approximate occurrence string graph whose label contains o_k, and $label(f(o_k))$ denote the label on node $f(o_k)$. Then the string obtained by eliminating duplicate terms from $label(f(o_1)) \cdots label(f(o_k))$ is a bounded estimate of $o_1 \cdots o_k$.

A limitation of the technique just described is its tendency to give pessimistic estimates of occurrence strings. A term $(a_1 \mid \cdots \mid a_\alpha)^+$ corresponds to the rather pessimistic assertion that any a_i in a set of mutually control-dependent points can be invoked after any other a_j. Straightforward adjustments to this algorithm give sharper estimates of occurrence strings when mutually dependent program points are executed in a fixed order. Assume, for example, that $\{ p_1, p_2, p_3 \}$ is a maximal set of mutually control-dependent

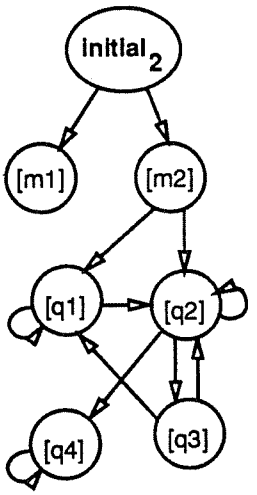
```
        procedure  main ();
          . . . .
[m1]     call P ();
          . . . .
[m2]     call Q ();
        end ;

        procedure  P ();
          . . . .
        end ;

        recursive  procedure  Q ();
[q1]     call  Q ();
          . . . .
[q2]     while
            . . . .
[q3]        call  Q ();
[q4]        while
               . . . .
            od
          od
        end ;

      endrec ;
```

Example program's update-site graph



Example program's approximate occurrence-string tree

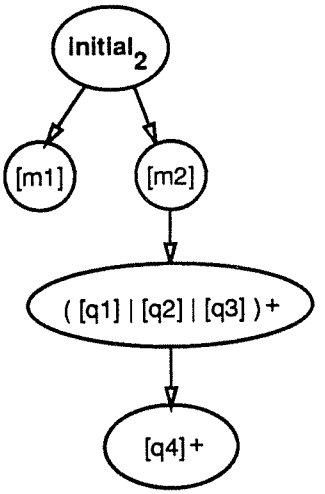


Figure 6.7. An example program, together with its unfolding-site graph and approximate-occurrence-string tree.

statements. Assume, furthermore, that "$[p_1]$ **call** $Q()$", "$[p_2]$ **call** $R()$", and "$[p_3]$ **call** $P()$" are points in procedures P, Q, and R, respectively, and that P, is always invoked before Q and R. Under these circumstances, it would be reasonable to use a term like $(p_1 p_2 p_3)^*\ (p_1 p_2 p_3 \mid p_1 p_2 \mid p_1)$ in an abstract occurrence string, instead of the less precise $(p_1 \mid p_2 \mid p_3)^+$.

6.4. Abstracting Sets of States

The techniques described in Sections 6.1 through 6.3 for abstracting occurrence strings, labeled stores, and sets of labeled stores can be used to generate bounded estimates of a program's set of states. In particular, ∇ must limit the number of distinct occurrence strings and labeled stores that the analysis pairs with every program point.

6.5. The Cost of Program Analysis

A careful analysis of the asymptotic complexity of using store graphs has been developed by Chase, Wegman, and Zadeck [Cha90]. These authors, in effect, argue that algorithms that use store graphs to estimate program behavior can have poor worst-case performance. Let ∇', for example, denote a stateset estimation operator that

* limits the number of structures in a store with a given allocation-point label, and
* merges every store that reaches each program point into one, representative store.

Roughly speaking, Chase, Wegman, and Zadeck argue that the worst-case running time of an analysis that

uses ∇' is $O(S^5)$, where S is the number of statements in a program P.[9] Chase *et. al.* use this argument to motivate what they refer to as an efficient algorithm for program analysis. This second algorithm computes *one* abstract store graph that represents *every* store might propagate to *every* point in a program. Chase *et. al.* argue, roughly speaking, that this algorithm either runs in time $O(S \log S)$, in time $O(S^4)$, or in time $O(S^7)$—depending on whether the number of references at a typical abstract structure is $O(1)$, $O(S)$, or $O(S^2)$, respectively. This paper also observes that the efficient algorithm would probably not work well for programs that contain procedures that allocate structures.

Other analyses of asymptotic complexity appear in theses by Chase, Ruggieri, and Hendren. Each of these authors use stateset estimation operators that merge all stores that reach every program point into a single store. Chase estimates a worst-case running time of $O(S^4)$ for his intraprocedural, label-driven analysis [Cha87]. Ruggieri, who k-limits stores, estimates a worst-case running time[10] of $O(S^2 \times V \times \alpha(S^2 \times V))$, where V is the number of identifier expressions of length $\leq k$ and α is the inverse Ackermann function [Rug87]. Hendren estimates a worst-case running time of $O(S^4 \times K^2)$ for interprocedural analysis, where K is the length of a program's longest identifier expression [Hen90].

Stransky's thesis presents empirical observations about the performance of his store-graph-based Lisp analyzer. The following two sentences, which are translated verbatim from Chapter 4 of [Str88], capture the overall tone of this discourse:

> The [heap-]graph is a disagreeable data structure to manipulate, and costly in space and in time of manipulation .... The algorithms for [heap-graph] manipulation present no strictly theoretical difficulties, but are all the same rather costly ....

The rest of the chapter contains an informal discussion of the costs of various operations on store graphs. Stransky singles out the divide-and-shrink operation and the handling of recursive function calls as particularly expensive operations.

Perhaps the most cogent observation that can be made about the expected cost of pointer program analysis is that there is still much work to be done in this area. The asymptotic estimates and anecdotal observations given here, while useful, do not adequately address the following, fundamental concerns about program analysis:

- Can a program P's form be used to determine the heuristics that should be used to analyze P?
- What is the cost of analyzing a "typical" program, according to a given heuristic?
- What improvement in quality of information (and increase in running time) might be expected from an increase in precision?
- Would some combination of heuristics work better than any one approach attempted to date?

This last remark is prompted by the observation that the various strategies for partitioning and reducing stores described in Section 6.1 work well for some program constructs, and poorly for others. This last remark is also prompted by Havlak's conjecture that it may be necessary to use easily discovered facts

---

[9] Complexity estimates in Section 6.5 are given, when possible, as a function of the number of statements that a program contains. These estimates, which are simplified versions of the ones given in the papers cited, were obtained by assuming that the number of allocation sites and variables that a program manipulates are both proportional to the number of statements that it contains.

[10] The estimate of asymptotic complexity given for Ruggieri's algorithm by Chase, Wegman, and Zadeck [Cha90] is the estimate that appears in equation (5.77) of Ruggieri's thesis; this (improved) estimate appeared in the thesis's concluding chapter.

about program evaluation to guide the discovery of harder ones [*private communication*]. It would be interesting to determine, for example, whether a multi-pass analysis that generated progressively refined its characterization of a program's aliases would prove more efficient than a comparable, one-pass analysis. These concerns, however, are matters for further study.

6.6. Other Related Work

A recent paper by Guarna describes a technique for alias analysis that discovers possible recurrences among a program's identifier expressions ([Gua90a]; see also [Gua90]). Guarna's technique for program analysis attempts to discover (*e.g.*) whether a program's identifier expressions satisfy identities such as "$x.hd = x.(tl.tl)^*.hd$" and "$x.tl^n = y.tl^n$". This algorithm uses statically gathered data about program evaluation, together with an algebra of selector expressions, to discover such identities. Guarna's algorithm for analyzing loops, for example, first determines what identifier expressions might be aliased after the first few iterations of a given loop L. The algorithm then uses the aforesaid algebra to derive new identities, and make conjectures about L's subsequent evaluation. The third step of the algorithm then attempts to verify the conjectured identities by rerunning the loop. There are pronounced similarities between Guarna's work and the conjectured use of graph grammars discussed in Section 6.1.2. The prinicipal limitation of Guarna's work (as presented in [Gua90a]) is the lack of a clear strategy for using rules to develop the desired recurrences.

The strategy presented in Section 6.3 for abstracting occurrence strings is related to strategies for abstracting stack configurations and states given by Harrison [Har89] and Deutsch [Deu90], respectively. Sharper estimates of a program's occurrence strings can be obtained by using (*e.g.*) superscripts of the form $ai + b$, where a and b are integers and i an integer variable. This technique for estimating occurrence strings is related to techniques that have been proposed for determining whether a iterations of a loop can be run in parallel [Goh90, Bod90].

7. DO DEPENDENCES CAPTURE A POINTER PROGRAM'S BEHAVIOR?

> *Although there exists an extensive body of work that makes use of program dependence graphs, we were unable to find any published proof that program dependence graphs were "adequate" as a program representation.... In this paper, we prove that for a language with assignment statements, conditional statements, and while-loops, a program dependence graph does capture a program's behavior.*
> —S. Horwitz, J. Prins, T. Reps [Hor87a]

Dependence analysis became popular in the 1970's, when Kuck, Muraoka, and Chen used a program's dependences to parallelize the execution of statements that manipulated arrays. Kuck *et. al.*, however, never gave a formal proof that their program-transformation techniques were sound. The use of informal arguments to justify dependence-based program transformations persisted until 1987, when Horwitz, Prins, and Reps showed the equivalence of programs with isomorphic control, flow, and def-order dependences, relative to a simple, structured language [Hor88]. Intuitively, this theorem justifies program transformations that permute a program's statements, but leave its dependences intact.

Since 1987, other theorems have been proved that justify the use of dependences to reason about programs (*cf* §7.5.2). None of these results, however, apply to languages that support reference variables, dynamic allocation, and procedures. The current chapter takes a first step towards justifying the use of dependences to represent programs in $\mathcal{H}$-like languages. Specifically, it shows that an analogue of Horwitz *et. al.*'s result also holds for programs in language $\mathcal{H}$.

Chapter 7 is divided into six sections.

Section 7.1 introduces the notion of a *dependence-based representation* (*dbr*). A *dbr* is a directed, labeled graph that depicts a subset of a program's dependences. Section 7.1 shows how operations on one type of *dbr*, the (Horwitz-Prins-Reps) *program dependence graph* (*pdg*), can be used to reason about a program's behavior.

Section 7.2 defines an example *dbr* for programs in language $\mathcal{H}$. This *dbr*, the *heap-language system dependence graph* (*hsdg*), is a variant of the *pdg* that supports procedures and reference variables.

Section 7.3 proves a theorem about the representational soundness of *hsdgs*. This theorem, the ***Pointer-Language Equivalence Theorem***, states that programs with isomorphic *hsdgs* map equivalent inputs to equivalent outputs. Since Section 7.3 is fairly dense and rather specialized, the casual reader would do well to read the sketch of the proof in the preface to Section 7.3, and skip the subsections.

Section 7.4 discusses the practical significance of the Pointer-Language Equivalence Theorem. Recall that the definition of language $\mathcal{H}$ (*cf.* Chapter 2) makes simplifying assumptions about freelists, procedure activation records, and atoms. The three subsections of Section 7.4 discuss how these assumptions affect the applicability of this result to real implementations of pointer languages. The most interesting subsection is probably 7.4.1, which shows that programs with isomorphic *hsdgs* may behave differently in the presence of a finite freelist.

Section 7.5 discusses related work. Section 7.5.1 surveys earlier *dbrs*. Section 7.5.2 reviews earlier soundness theorems for *dbrs*.

Section 7.6 critiques the *hsdg*. The *hsdg* fails to incorporate three recent ideas for improving *dbrs*. The first idea is that *dbrs* should be designed without def-order (and output) dependences (*cf.* §7.5.1.3). The

second is that different vertices should be used to represent distinct values in the initial and final stores (*cf.* §7.5.1.5). The third is that every *interprocedural* dependence should have one endpoint at a program point that implements a call statement, and the other endpoint at a program point that implements a callee's entry (or exit) code (*cf* §7.5.1.2). Section 7.6 explains why the presence of reference variables and dynamic allocation in language $\mathcal{H}$ makes these goals hard to satisfy. Section 7.6 also discusses improvements to the *hsdg* that *might* address these concerns—improvements, however, that require further research.

7.1. The Use of Dependence-Based Representations in Program Analysis

Algorithms that use dependences to model program behavior often organize a program's dependences as a graph. A graph that models a program P's behavior generally contains one vertex for each of P's points of control, and one edge for each of P's "interesting" dependences. For example, if V_p is a vertex that corresponds to point p, and V_q a vertex that denotes a point q, then the edge (V_p, V_q) corresponds to a dependence $p \longrightarrow q$. Edges may be labeled with values that characterize a program's dependences. An edge, for example, that corresponds to a dependence d may be labeled with a value that gives d's type, or its distance, or the carriers of d—or all of the above. This organization is convenient because it allows certain key assertions about program behavior to be phrased in terms of operations on graphs.

Dependence-depicting graphs differ according to the types of languages they support, the types of dependences they depict, and the types of labels they contain. In this thesis, an arbitrary dependence-depicting graph will be referred to as a *dependence-based representation* (*dbr*). This term has been chosen for historic reasons; more natural terms like "program graph", "program dependence graph", and "program representation graph" have already appropriated for specific types of *dbrs* (*cf.* §7.5.1).

The types of applications that a *dbr* can support vary according to the information it contains. The *dbr* depicted in Figure 7.1, for example, has been used to characterize program behavior w.r.t. a simple, structured language that lacks procedures, reference variables, arrays, and structures. This *dbr*, the Horwitz-Prins-Reps ***program dependence graph*** (***pdg***), contains one edge for every flow, control, and def-order dependence that the program exhibits. Every edge e in this *dbr* can have up to three sets of labels: one label that identifies the type of d; a second set of labels that identifies the loops that carry d (if d is a flow dependence); and a third label that identifies the points that witness d (if d is a def-order dependence).

Horwitz, Prins, Reps, and Yang proved that the HPR *pdg* characterizes several important aspects of a program's behavior (*cf.* §7.5.2). Horwitz, Prins, and Reps, for example, showed that a program transformation that leaves P's *pdg* intact preserves P's meaning [Hor88]. This result implies that the following program is equivalent to the one shown in Figure 7.1:

```
[5] x := g(); [6] if x ?= 0 then [7] x := 1 fi; [8] z := 1/x;
[1] x := f(); [2] if x ?= 0 then [3] x := 1 fi; [4] y := 1/x;
[9] print(y+z)
```

Reps and Yang also proved that the HPR *pdg* can be used to determine the set of statements that might affect—or be affected by—the sequence of values computed at a point p. The algorithm for determining a program P's logically related statements, which was first proposed by Ottenstein and Ottenstein [Ott84], is illustrated in Figures 7.2 and 7.3. To find the statements that might affect the evaluation of a statement s, one computes the ***backward slice*** of a *pdg* G w.r.t. s: *i.e.*, the set of paths π_b that start at G's start vertex and end at s (*cf.* Figure 7.2). Any point that does not lie on a path in π_b cannot affect the sequence of values

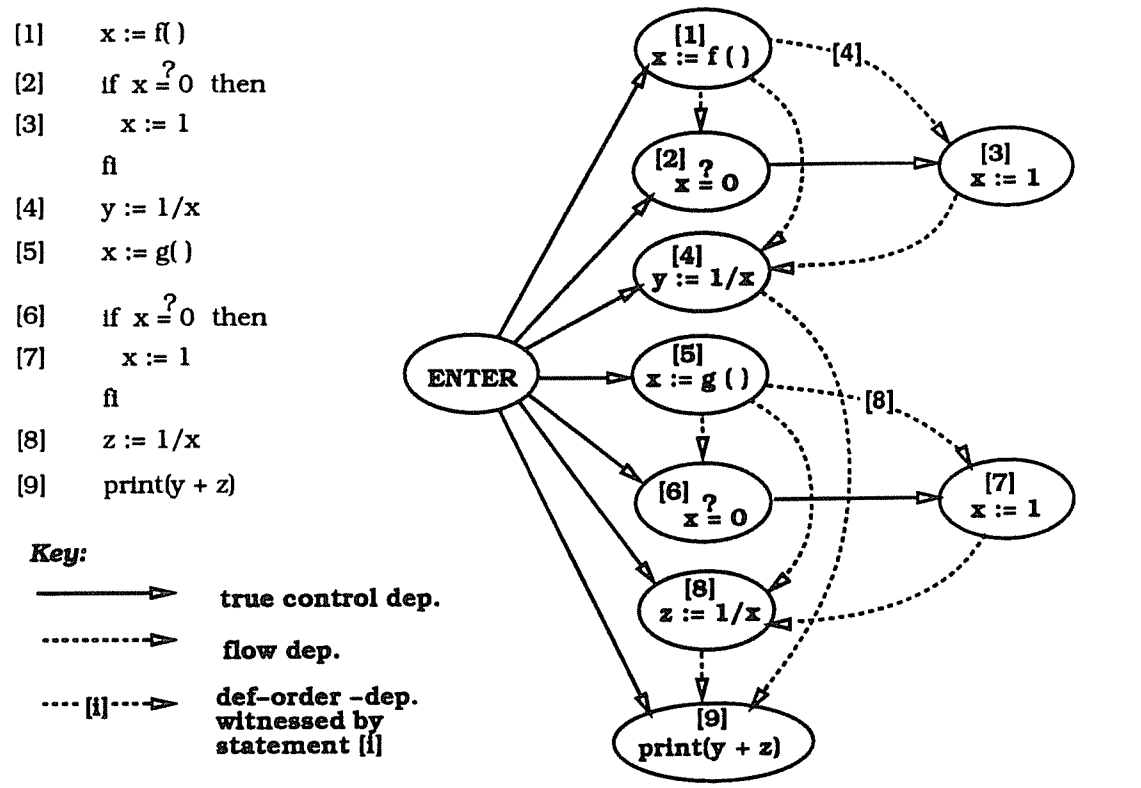


Figure 7.1. A Horwitz-Prins-Reps *pdg*.

computed at *s*. To find the statements that *s* might affect, one computes the ***forward slice*** of a *pdg* G w.r.t. *s*: *i.e.*, the set of paths π_f that start at *s* and end in G (*cf.* Figure 7.3). Statement *s* cannot affect the sequence of values computed at any point that does not lie on a path in π_f. (*N.B.*: there is one trivial exception to these observations; any statement can affect the evaluation of any other statement by causing P to fail.)

Soundness proofs like the ones given by Horwitz *et. al.* are important, since not all *dbrs* are suitable for reasoning about the same types of program behavior. The HPR *pdg*, for example, cannot be used to identify which of a program's statements can be evaluated in parallel. Soundness proofs are important for a second reason: experience has shown that *dbrs* must sometimes capture subtle information about program behavior. Horwitz, Prins, and Reps, for example, discovered that information about a dependence's status was needed to ensure that the following (inequivalent) programs were represented by different *pdgs*:

```
[1] x := 0                          [1] x := 0
[2] while pred1 do                  [2] while pred1 do
[3]   y := x                        [4]   if pred2 then [5] x := 1 fi
[4]   if pred2 then [5] x := 1 fi   [3]   y := x
    od                                  od
```

The dependence [5] $\rightarrow_f$ [3] is *both* carried by, and independent of, the loop at [2] in the right-hand program. The observation that [5] $\rightarrow_f$ [3] is *not* independent of the loop at [2] in the left-hand program is important, since it suggests that the two programs may have different meanings.

A detailed discussion of other important *dbr*-based program transformations is beyond the scope of this thesis. This includes program integration [Rep89], vectorization [Bax89], loop interchange [All87], and other optimizations discussed in Chapter 3.

7.2. A *Dbr* for Language $\mathcal{H}$

The ***heap-language system dependence graph***, whose definition is given below, is a new type of *dbr* for language $\mathcal{H}$.

DEFINITION. A ***heap-language system dependence graph*** (*hsdg*) is a labeled, directed graph that depicts a pointer program's control, flow, and def-order dependences w.r.t. a set of inputs. Specifically, let P be a pointer program, and *InSet* a set of inputs. Let S_P be an *hsdg* that represents P w.r.t. *InSet*. Then S_P **must** contain:

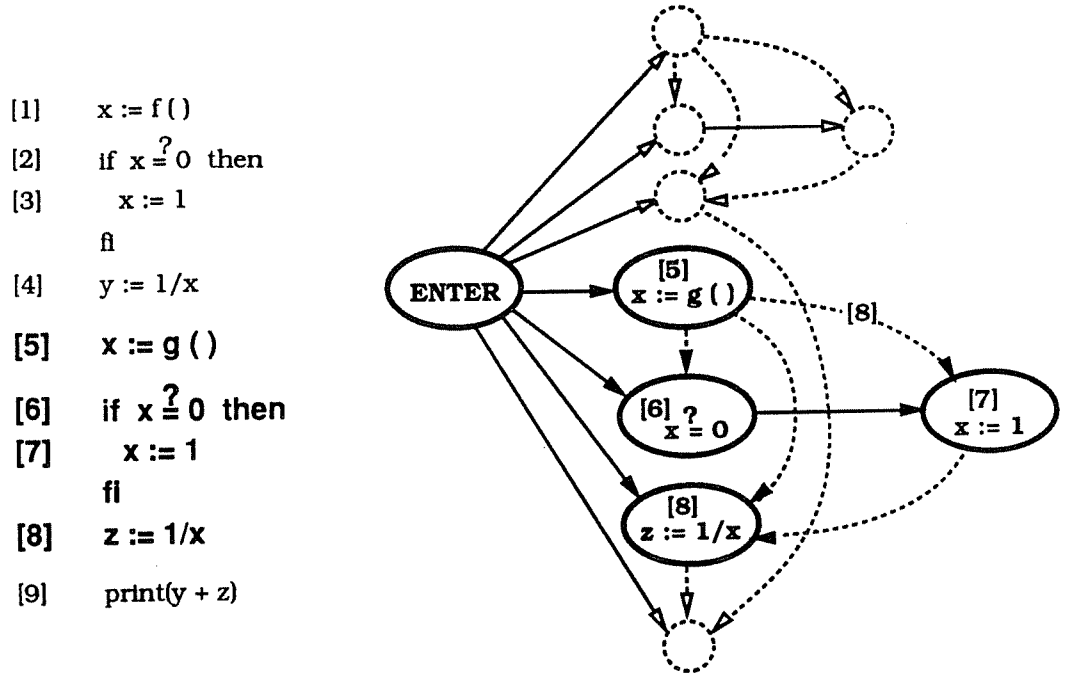


Figure 7.2. The backward slice of the example program w.r.t. statement [8]. The statements in the slice, highlighted in boldface, are those statements s such that there is a path from ENTRY to s to [8].

```
[1]   x := f()
[2]   if x ?= 0 then
[3]     x := 1
      fi
[4]   y := 1/x
[5]   x := g()
[6]   if x ?= 0 then
[7]     x := 1
      fi
[8]   z := 1/x
[9]   print(y + z)
```

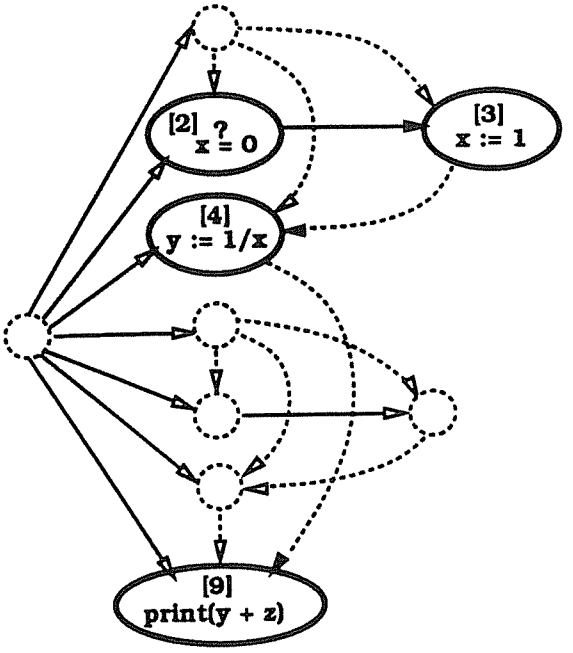


Figure 7.3. The forward slice of the example program w.r.t. statement [2]. The statements in the slice, highlighted in boldface, are those statements *s* such that there is a path from [2] to *s*.

- One vertex for each of *P*'s program points (*cf.* § 3.2.1), **initial**$_2$ and **final** excepted. Vertices that represent predicates and assignment statements are labeled with the name of the point that they represent. The following labels are assigned to vertices that represent special program points:
  - The vertex that represents the program point that initializes the store is labeled **initial**$_1$.
  - A vertex that represents a point that passes a return address to a callee is labeled δ_0 := <*retadr*>.
  - A vertex that defines a call site's *i*th actual parameter, a_i, is labeled $\delta_i := a_i$.
  - The vertex that represents procedure A's entry point is labeled *enter A*.
  - The vertices that represent the three points that initialize A's local environment are labeled *_temp._prev* := *_local*, *_local* := *_temp*, and *_local._callctxt* := *_local._prev.*δ_0.
  - The vertex that represents the point that initializes A's *i*th formal parameter, f_i, is labeled $f_i := \delta_i$.
- One edge for each of *P*'s control, flow, and def-order dependences. Specifically:
  - One edge for each of *P*'s interprocedural control dependences, **initial**$_2$ $\rightarrow_c$ *main*() excepted.
  - One edge for each of *P*'s intraprocedural control dependences (*cf.* §3.2.2). Edges that represent true- and false-valued control dependences are labeled **true** and **false**, respectively.
  - One edge for each of *P*'s flow and def-order dependences w.r.t. *InSet* (*cf.* § 3.2.2). Every flow and def-order dependence *d* is labeled with the call and loop sites that carry *d* (*cf.* § 3.3).

Graph S_P **may** also contain edges that correspond to ***spurious dependences***: syntactically possible, but non-existent, flow and def-order dependences in P. S_P may contain an edge $p \longrightarrow_f q$ if there is a path from p to q in P's control-flow graph. S_P may also contain an edge $p \longrightarrow_{do(r)} q$ if S_P contains $p \longrightarrow_f r$ and $q \longrightarrow_f r$. These edges are a concession to the limitations of program analysis; it is impossible, in general, to determine a program's exact data dependences (*cf.* § 3.4). □

The *hsdg* is similar to another *dbr* for programs with procedures, the *system dependence graph* (*sdg*) [Hor88a, Hor90a]. The *sdg*, which is described in Section 7.5.1.5, is an extension of the *pdg* that supports languages with procedures. Roughly speaking, the *hsdg* differs from the *sdg* in the following three ways:

- Each program in the model language supported by *sdgs* has exactly one *sdg*. This *sdg* depicts an approximation to a program P's flow and def-order dependences that is computed from P's control-flow graph. Pointer programs, on the other hand, can have more than one *hsdg*. A program P's *hsdg* varies according to how P's dependences are computed:

  * An *hsdg* that depicts P's dependences w.r.t. a set of input stores *InSet* may differ from one that depicts P's dependences w.r.t. an *InSet'* that differs from *InSet*.

  * *Hsdgs* also vary according to how many spurious dependences they depict. For example, let H_1 and H_2 be *hsdg* for a program P w.r.t. the set of input stores *InSet*. H_1 may depict a spurious dependence $p \longrightarrow_d q$ that H_2 omits. (H_2 provides a more precise characterization of P's behavior; H_1, on the other hand, might be easier to compute).

- The languages that *hsdgs* and *sdgs* represent make different assumptions about procedure evaluation. *Hsdgs* and *sdgs* therefore have different procedure initialization and finalization vertices. *Sdgs*, in effect, give a cleaner characterization of the interface between caller and callee than *hsdgs*.

- *Hsdgs* contains exactly one vertex, $\mathbf{initial}_1$, that models the initial state of a program's store. *Sdgs* may contain multiple vertices that depict the initial definitions, and final uses, of specific variables in the store. *Sdgs*, in effect, give a cleaner characterization of a program's initial and final states.

Reasons why an *sdg*-like characterization of procedure calls, initial variable definitions, and final variable uses are harder to obtain for language $\mathcal{H}$ are given in Section 7.6.

An important feature of the *hsdg* is the lack of edges that depict data dependences that arise through the freelist. Such dependences are omitted from *hsdgs* because they represent needless constraints on program execution. Consider, for example, the following example program P:

```
[p] a := new( ··· )     /*** a and b are not aliased before the evaluation of [p]
[q] b := new( ··· )
```

Program P exhibits a freelist-mediated dependence $p \longrightarrow_{stream} q$. Let Q be the program obtained by reversing p and q:

```
[q] b := new( ··· )     /*** a and b are not aliased before the evaluation of [p]
[p] a := new( ··· )
```

Clearly, programs P and Q generate indistinguishable final stores, up to how locations are paired with the structures allocated at p and q. The dependence $p \longrightarrow_{stream} q$, in effect, is a constraint on the order in which locations are removed from P's freelist. Such constraints, however, have no significant affect on

programs in referentially transparent languages.

This claim that freelist-mediated dependences may be removed from *hsdgs* without compromising the resulting characterization of program behavior is argued in Section 7.3.

7.3. A Basis for Reasoning about Pointer-Language Programs

A recent report by Pfeiffer and Selke takes a first step towards showing that *dbrs* can characterize the semantics of pointer programs [Pfe91, Pfe91a]. Pfeiffer and Selke first define a *dbr* for a procedureless, type-declaration-free subset of $\mathcal{H}$. They then show that this *dbr*, the *hpdg*, captures certain facets of program behavior. Specifically, they show that two programs with isomorphic *hpdgs* have equivalent behavior. They also show that the *hpdg* gives a sound characterization of a program's slices.

Section 7.3 continues the work started in [Pfe91, Pfe91a]. This section's principal theorem, the ***Pointer-Language Equivalence Theorem***, demonstrates that two programs $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ with isomorphic *hsdgs* exhibit *equivalent behavior* when applied to a store $\sigma_{\mathcal{H}}$. Intuitively, computations $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are said to exhibit equivalent behavior iff either

- $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ fail to terminate successfully, or
- $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ compute *equivalent* final stores, and there is an isomorphism between the points of $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ such that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ generate corresponding sequences of values at isomorphic program points.

The notion of ***store equivalence*** captured in the Pointer-Language Equivalence Theorem is the standard notion of store equivalence w.r.t. a referentially transparent language. Intuitively, stores $\sigma_{\mathcal{H}}$ and $\tau_{\mathcal{H}}$ are equivalent if the accessible portions of $\sigma_{\mathcal{H}}$ and $\tau_{\mathcal{H}}$ are isomorphic, up to how structures are paired with locations.

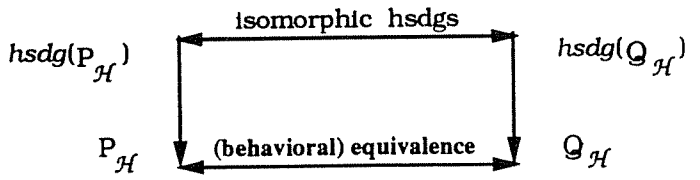
DEFINITION. Let $\sigma_{\mathcal{H}}$ be a member of $Store_{\mathcal{H}}$, and *gEnv* be σ's global environment. Let *Idexp* be the set of *all* path expressions in language $\mathcal{H}$, including path expressions (such as *_local._prev.a*) that are accessible only to the implementation. An $idexp \in Idexp$ ***denotes*** a structure s in $\sigma_{\mathcal{H}}$ if $selexp(\sigma, gEnv, idexp) = s$. (*N.B.*: *selexp* is defined in Appendix 1.) □

DEFINITION. Let $\sigma_{\mathcal{H}} \in Store_{\mathcal{H}}$ be a store, and *gEnv* be σ's global environment. Let *Idexp* be the set of *all* path expressions in language $\mathcal{H}$. Two identifier expressions $idexp \in Idexp$ and $idexp' \in Idexp$ are ***aliased*** w.r.t. $\sigma_{\mathcal{H}}$ if $selexp(\sigma, gEnv, idexp) = selexp(\sigma, gEnv, idexp')$. □

DEFINITION (***equivalent stores in language*** $\mathcal{H}$). Let $\sigma_{\mathcal{H}}$ and $\tau_{\mathcal{H}}$ be members of $Store_{\mathcal{H}}$. Let *Idexp* be the set of *all* path expressions, including path expressions (such as *_local._prev.a*) that are accessible only to the implementation. Stores $\sigma_{\mathcal{H}}$ and $\tau_{\mathcal{H}}$ are ***equivalent***, written $\sigma_{\mathcal{H}} \approx_{\mathcal{H}} \tau_{\mathcal{H}}$, iff

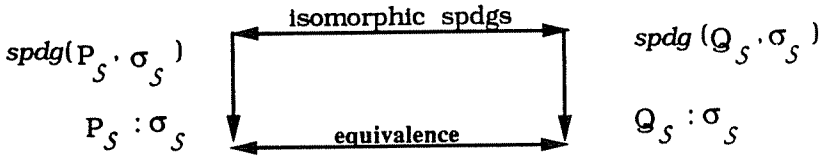
- for all $idexp \in Idexp$, *idexp* denotes a structure of type t w.r.t. $\sigma_{\mathcal{H}}$ iff *idexp* denotes a structure of type t w.r.t. $\tau_{\mathcal{H}}$;
- for all $idexp \in Idexp$, *idexp* denotes an atom w.r.t. $\sigma_{\mathcal{H}}$ whose value is v iff *idexp* denotes an atom w.r.t. $\tau_{\mathcal{H}}$ whose value is v; and
- for all $idexp \in Idexp$ and $idexp' \in Idexp$, *idexp* and *idexp'* are aliased w.r.t. $\sigma_{\mathcal{H}}$ iff *idexp* and *idexp'* are aliased w.r.t. $\tau_{\mathcal{H}}$. □

The following is a pictorial representation of the Pointer-Language Equivalence Theorem.



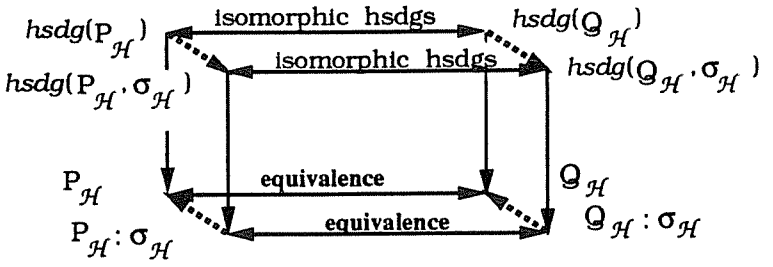
Intuitively, this figure asserts that the behavioral equivalence of two programs $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ can be established by establishing that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs*.

The Pointer-Language Equivalence Theorem is proved by reducing it to a second equivalence theorem. This second theorem, the ***Spdg Equivalence Theorem***, characterizes the representational soundness of a *dbr* for a second, simpler language. Roughly speaking, this second language $\mathcal{S}$ is a subset of $\mathcal{H}$ that lacks dynamic allocation, procedures, loops, dereferencing, and aggregate variables. This second *dbr*, the ***spdg***, is similar to the Horwitz-Prins-Reps (HPR) *pdg* described in Section 7.1. The principal difference between the HPR *pdg* and the *spdg* is that the *spdg*, like the *hsdg*, portrays a more dynamic notion of data dependence: *i.e.*, one that reflects a program's possible executions. An HPR *pdg* for program P *must* contain the edge $p \longrightarrow_d q$ when P's control-flow graph contains paths from p to q that satisfy certain criteria—even if statements along these paths never evaluate. An *spdg* for P, on the other hand, may omit $p \longrightarrow_d q$ if none of P's *evaluations* exhibit $p \longrightarrow_d q$. The *Spdg Equivalence Theorem*, roughly speaking, states that two language S programs with isomorphic *spdgs* are behaviorally equivalent programs.



The proof of the Pointer-Language Equivalence Theorem takes the form of a seven-step reduction. The first five steps in the proof reduce the statement of the Pointer-Language Equivalence Theorem to an equivalent assertion about loop-free, procedure-free programs in $\mathcal{H}$. The final two steps of the proof reduce this latter assertion to the *Spdg* Equivalence Theorem. This proof is presented in detail in the six subsections of Section 7.3. A sketch of the argument now follows.

Step 1. The Pointer-Language Equivalence Theorem asserts that programs $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ exhibit equivalent behavior when (1) $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. a set of stores *InSet*, and (2) $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are run on an arbitrary $\sigma_{\mathcal{H}} \in$ *InSet*. Step 1 specializes the statement of the theorem to an equivalent assertion about a *specific* pair of computations. An equivalence relation is first defined on computations in language $\mathcal{H}$. It is then argued that the Pointer-Language Equivalence Theorem holds if ($\bullet$1) $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent for all $\sigma_{\mathcal{H}} \in$ *InSet* such that $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ terminates.



Step 2, 3, and 4 reduce assertion (●1) to a comparable assertion about flattened approximations to $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$.

DEFINITION (***flattened program (language*** $\mathcal{H}$ **)**). Let $P_{\mathcal{H}}$ be a program in language $\mathcal{H}$. Program $P_{\mathcal{H}}$ is *flattened* iff $P_{\mathcal{H}}$ consists of one procedure, *main*(); $P_{\mathcal{H}}$ contains no **call** and no **while** statements; and every conditional statement in $P_{\mathcal{H}}$ is of one of two forms:

* "**if** *pred* **then fail else** ··· **fi**"
* "**if** *pred* **then** ··· **else fail fi**" □

Intuitively, Steps 2 through 4 create simple, finite approximations to $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$—approximations that, nevertheless, are behaviorally equivalent to $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ **w.r.t.** $\sigma_{\mathcal{H}}$. This simplification of $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ is made possible by the hypothesis that $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ terminates. The principal reason for flattening $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ is that this transformation makes it easier to argue that freelist-mediated dependences represent needless constraints on program evaluation (*cf.* Step 7).

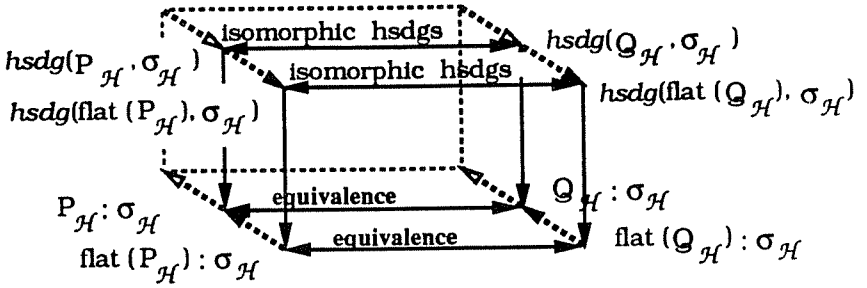
Step 2 reduces assertion (●1) to a comparable assertion about programs that lack procedure calls. This reduction is performed in two stages. The trace of $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ is first used to in-line expand certain procedure calls in $P_{\mathcal{H}}$, and to replace others by **fail** statements. This reduction yields a second computation, $\mathit{flat}(P_{\mathcal{H}}):\sigma_{\mathcal{H}}$, that is (*) closely related to $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$. In particular, an auxiliary procedure $A()$ is evaluated at a specific point in $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ *iff* an in-line-expanded version of $A()$ is evaluated at a corresponding point in $\mathit{flat}(P_{\mathcal{H}}):\sigma_{\mathcal{H}}$. Throughout Section 7.3, pairs of closely related computations like $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $\mathit{flat}(P_{\mathcal{H}}):\sigma_{\mathcal{H}}$ are said to be ***congruent***. Specifically, computations c and $\mathit{reduced}(c)$ are deemed congruent whenever (1) $\mathit{reduced}(c)$ is produced by transforming c, and (2) a close correspondence between $\mathit{reduced}(c)$ and c makes it possible to use one computation to reason about the other.

The second stage of the reduction uses the trace of $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ to in-line expand certain procedure calls in $Q_{\mathcal{H}}$, and to replace others by **fail** statements. Let $\mathit{flat}(Q_{\mathcal{H}})$ denote the reduced $Q_{\mathcal{H}}$. The definitions of equivalence and congruence now imply that (●1) $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}:\sigma_{\mathcal{H}}$ are equivalent if (*) $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $\mathit{flat}(P_{\mathcal{H}}):\sigma_{\mathcal{H}}$ are congruent; (**) $\mathit{flat}(Q_{\mathcal{H}}):\sigma_{\mathcal{H}}$ and $\mathit{flat}(P_{\mathcal{H}}):\sigma_{\mathcal{H}}$ are equivalent; and (***) $Q_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $\mathit{flat}(Q_{\mathcal{H}}):\sigma_{\mathcal{H}}$ are congruent. Claim (*) follows immediately from the reduction (see above). Claim (**) follows from Step 3 (below) and a second argument that $\mathit{flat}(P_{\mathcal{H}})$ and $\mathit{flat}(Q_{\mathcal{H}})$ have isomorphic *hsdgs*. Claim (***) is demonstrated by showing that the second stage of the reduction preserves the semantics of $Q_{\mathcal{H}}:\sigma_{\mathcal{H}}$: *i.e.*, that the reduction does not (*e.g.*) replace an evaluating *call* statement in $Q_{\mathcal{H}}$ with a **fail** statement. In particular, assertions (*), (**), and the close correspondence between $Q_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $\mathit{flat}(Q_{\mathcal{H}}):\sigma_{\mathcal{H}}$ are used to show the correctness of the reduction.

To summarize this somewhat complicated argument, the reduction performed at Step 2 yields two programs, *flat* $(P_{\mathcal{H}})$ and *flat* $(Q_{\mathcal{H}})$, that satisfy the following three assertions:

- $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and *flat* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are congruent computations.
- $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and *flat* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are congruent computations whenever *flat* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flat* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are equivalent computations.
- *flat* $(P_{\mathcal{H}})$ and *flat* $(Q_{\mathcal{H}})$ have isomorphic *hsdgs* w.r.t. $\sigma_{\mathcal{H}}$.

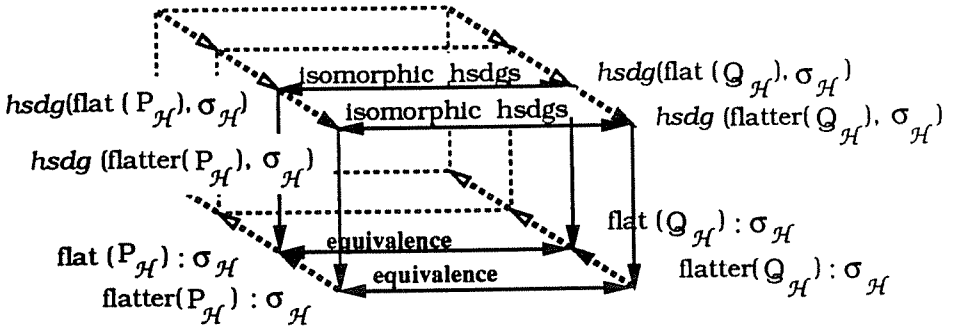
Step 2, in effect, reduces assertion (•1) to the claim that (•2) the equivalence of *flat* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flat* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ is implied by the isomorphism of *flat* $(P_{\mathcal{H}})$'s and *flat* $(Q_{\mathcal{H}})$'s *hsdgs*.



Step 3 reduces assertion (•2) to a comparable assertion about programs that lack loops. The trace of *flat* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ is used to replace loops in *flat* $(P_{\mathcal{H}})$ and *flat* $(Q_{\mathcal{H}})$ with nested if-then-else statements. This reduction yields two programs, *flatter* $(P_{\mathcal{H}})$ and *flatter* $(Q_{\mathcal{H}})$, that satisfy the following three assertions:

- *flat* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flatter* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are congruent computations.
- *flat* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flatter* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are congruent computations whenever *flatter* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flatter* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are equivalent computations.
- *flatter* $(P_{\mathcal{H}})$ and *flatter* $(Q_{\mathcal{H}})$ have isomorphic *hsdgs* w.r.t. $\sigma_{\mathcal{H}}$.

Step 3, in effect, reduces assertion (•2) to the claim that (•3) the equivalence of *flatter* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flatter* $(Q_{\mathcal{H}}): \sigma_{\mathcal{H}}$ is implied by the isomorphism of *flatter* $(P_{\mathcal{H}})$'s and *flatter* $(Q_{\mathcal{H}})$'s *hsdgs*.

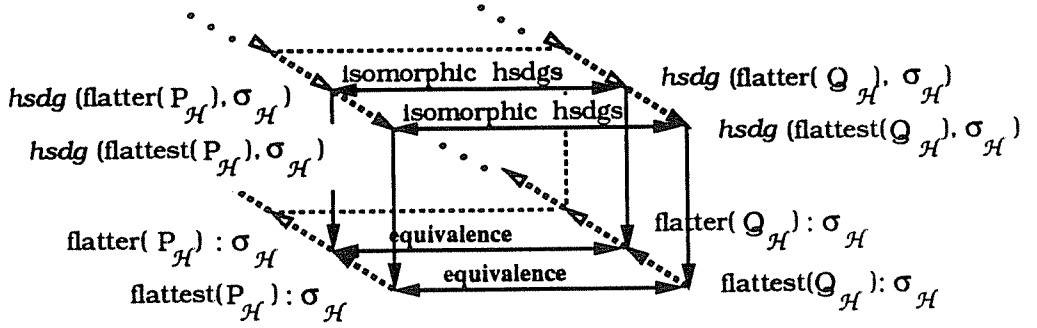


Step 4 reduces assertion (•3) to a comparable assertion about programs that have exactly one valid execution path. The trace of *flatter* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ is used to replace conditionals in *flatter* $(P_{\mathcal{H}})$ and *flatter* $(Q_{\mathcal{H}})$ with conditionals that have at most one valid consequent. This reduction yields two programs, *flattest* $(P_{\mathcal{H}})$ and *flattest* $(Q_{\mathcal{H}})$, that satisfy the following three assertions:

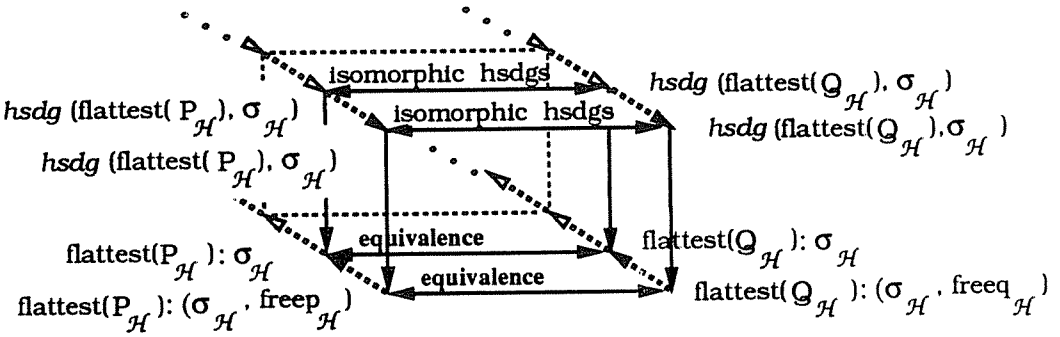
- *flatter* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ and *flattest* $(P_{\mathcal{H}}): \sigma_{\mathcal{H}}$ are congruent computations.

- *flatter* $(Q_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ and *flattest* $(Q_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ are congruent computations whenever *flattest* $(P_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ and *flattest* $(Q_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ are equivalent computations.
- *flattest* $(P_{\mathcal{H}})$ and *flattest* $(Q_{\mathcal{H}})$ have isomorphic *hsdgs* w.r.t. $\sigma_{\mathcal{H}}$.

Step 4, in effect, reduces assertion (•3) to the claim that (•4) the equivalence of *flattest* $(P_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ and *flattest* $(Q_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ is implied by the isomorphism of *flatter* $(P_{\mathcal{H}})$'s and *flatter* $(Q_{\mathcal{H}})$'s *hsdgs*.



Step 5 specializes assertion 4 to a pair of freelists, *freep* $_{\mathcal{H}}$ and *freeq* $_{\mathcal{H}}$. Let the expression *flattest* $(P_{\mathcal{H}})$: $(\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ denote the result of evaluating *flattest* $(P_{\mathcal{H}}) : \sigma_{\mathcal{H}}$ w.r.t. the freelist *freep* $_{\mathcal{H}}$. Step 5 shows that assertion (•4) is equivalent to the claim that (•5) the equivalence of *flattest* $(P_{\mathcal{H}})$: $(\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ and *flattest* $(Q_{\mathcal{H}})$: $(\sigma_{\mathcal{H}}, freeq_{\mathcal{H}})$ is implied by the isomorphism of *flattest* $(P_{\mathcal{H}})$'s and *flattest* $(Q_{\mathcal{H}})$'s *hsdgs*.



Here, *freep* $_{\mathcal{H}}$ is an arbitrary freelist, and *freeq* $_{\mathcal{H}}$ is a specific permutation of *freep* $_{\mathcal{H}}$ whose form depends on the form of *flattest* $(P_{\mathcal{H}})$.

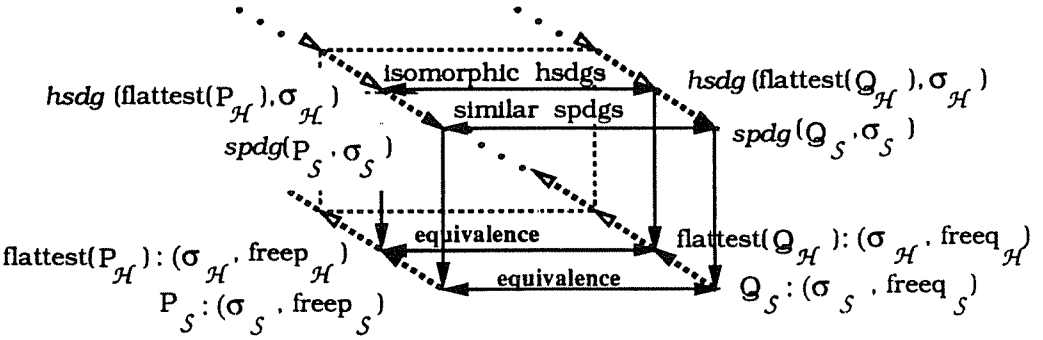
The reason for fixing *freep* $_{\mathcal{H}}$ and *freeq* $_{\mathcal{H}}$ before performing the translation to language $\mathcal{S}$ (Step 6) is that the specified choice of freelists simplifies Step 7. Specifically, let *computation* $_P$ and *computation* $_Q$ denote the reduced versions of *flattest* $(P_{\mathcal{H}})$: $(\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ and *flattest* $(Q_{\mathcal{H}})$: $(\sigma_{\mathcal{H}}, freeq_{\mathcal{H}})$ generated by Step 6. The assumptions imposed on *freep* $_{\mathcal{H}}$ and *freeq* $_{\mathcal{H}}$ in Step 5 ensure that corresponding evaluations of the simulated **new()** operator in *computation* $_P$ and *computation* $_Q$ return the *same* simulated address. This close correspondence between *computation* $_P$'s and *computation* $_Q$'s simulated allocation operations simplifies the proof that *computation* $_P$ and *computation* $_Q$ are equivalent. In particular, this correspondence ensures that the reduction described in Step 7 yields programs with isomorphic *spdgs*. This, in turn, allows the *Spdg* Equivalence Theorem—the theorem upon which the whole argument rests—to be applied in a straightforward manner.

Step 6 translates the computations in Step 5 into language $\mathcal{S}$. Let $P_{\mathcal{S}}$ and $Q_{\mathcal{S}}$ denote the translated versions of *flatten*$(P_{\mathcal{H}})$ and *flatten*$(Q_{\mathcal{H}})$; $\sigma_{\mathcal{S}}$, the translated version of $\sigma_{\mathcal{H}}$; and *freep*$_{\mathcal{S}}$ and *freeq*$_{\mathcal{S}}$, the translated *freep*$_{\mathcal{H}}$ and *freeq*$_{\mathcal{H}}$. The translated objects created in Step 6 satisfy the following three assertions:

- *flattest*$(P_{\mathcal{H}}) : (\sigma_{\mathcal{H}}, \textit{freep}_{\mathcal{H}})$ and $P_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freep}_{\mathcal{S}})$ are congruent computations.
- *flattest*$(Q_{\mathcal{H}}) : (\sigma_{\mathcal{H}}, \textit{freeq}_{\mathcal{H}})$ and $Q_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freeq}_{\mathcal{S}})$ are congruent computations whenever $P_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freep}_{\mathcal{S}})$ and $Q_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freeq}_{\mathcal{S}})$ are equivalent computations.
- $P_{\mathcal{S}}$ and $Q_{\mathcal{S}}$ have ***similar*** *spdgs* w.r.t. $\sigma_{\mathcal{S}}$.

DEFINITION (***similar spdgs***). Two *spdgs* are *similar* if they are isomorphic up to edges that represent freelist-mediated data dependences. □

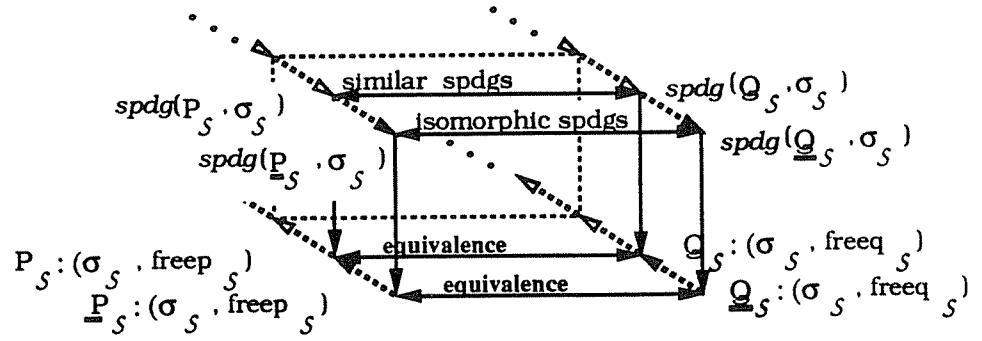
Step 6, in effect, shows that assertion (•5) is equivalent to the claim that (•6) the equivalence of $P_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freep}_{\mathcal{S}})$ and $Q_{\mathcal{S}} : (\sigma_{\mathcal{S}}, \textit{freeq}_{\mathcal{S}})$ is implied by the similarity of $P_{\mathcal{S}}$'s and $Q_{\mathcal{S}}$'s *spdgs* and the form of *freep*$_{\mathcal{S}}$ and *freeq*$_{\mathcal{S}}$.



Step 7 completes the reduction. A dependence-breaking program transformation that substitutes constants for accesses of the simulated freelist reduces $P_{\mathcal{S}}$ and $Q_{\mathcal{S}}$ to two related programs, $\underline{P}_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}}$. Programs $\underline{P}_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}}$ are then shown to satisfy the following three assertions:

- $P_{\mathcal{S}} : \sigma_{\mathcal{S}}$ and $\underline{P}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ are congruent computations.
- $Q_{\mathcal{S}} : \sigma_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ are congruent computations whenever $\underline{P}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ are equivalent computations.
- $\underline{P}_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}}$ have isomorphic *spdgs* w.r.t. $\sigma_{\mathcal{S}}$.

Step 7, in effect, reduces assertion (•6) to the claim that (•7) the equivalence of $\underline{P}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ and $\underline{Q}_{\mathcal{S}} : \sigma_{\mathcal{S}}$ is implied by the isomorphism of $\underline{P}_{\mathcal{S}}$'s and $\underline{Q}_{\mathcal{S}}$'s *spdgs*.



The observation that assertion (•7) is true by the *Spdg* Equivalence Theorem now completes the proof of the Pointer-Language Equivalence Theorem.

This completes the sketch of the Pointer-Language Equivalence Theorem. The detailed presentation given below differs from this sketch in the following two ways. The two sections that follow the current section, Sections 7.3.1 and 7.3.2, first explain the technique used to simulate language *H* computations in language *S*. The remaining sections then present the seven stages of the proof in a bottom-up (rather than top-down) fashion: this simplifies the presentation by allowing successive sections to build on theorems proved in previous sections.

7.3.1. Language *S*

Language *S*, whose concrete syntax is shown in Figure 7.4, is a procedure-less, loop-less, goto-less language that supports scalar variables and conditionals. It lacks reference variables, structures, and

Program	→ *Stmt_list*	*Cond*	→ VAR **is** TYPE
Stmt_list	→ *Stmt* {; *Stmt*}*		→ *SimpleExp* > *SimpleExp*
Stmt	→ **if** *Cond* **then** *Stmt_list* **else** *Stmt_list* **fi**		→ *SimpleExp* $\stackrel{?}{=}$ *SimpleExp*
	→ **case** *Switch* **in** {REF : *Stmt_List*}* **esac**		→ *SimpleExp* < *SimpleExp*
	→ **assert** *Cond*		→ **not** *Cond*
	→ VAR := *Exp*	*Switch*	→ VAR | **freelist**( ) | REF
	→ **fail**	*Exp*	→ PRIMFN (*SimpleExp*, ··· ,*SimpleExp*)
	→ **skip**		→ *SimpleExp*
		SimpleExp	→ VAR | ATOM | REF
		REF	→ **&0** | **&1** | ··· | *undefined*

TYPE is a set of type designators.
VAR is a set of alphanumeric variable names.
ATOM is a set of atomic values. Elements of REF simulate references to structures. ATOM and REF are disjoint.
PRIMFN is a set of primitive functions.

Figure 7.4. The concrete syntax for language *S*.

dynamic allocation. Appendix 7 gives an operational semantics for language $\mathcal{S}$. A sketch of this semantics now follows.

Language $\mathcal{S}$'s meaning function, $\mathbf{M}_{\mathcal{S}}$, defines a program as a map from a (store,input-stream) pair to a store. A *store* is a collection of variables. The input stream, hereafter called the *freelist*, is a list of simulated references (*i.e.*, elements of REF).

Elements of PRIMFN are self-contained, referentially transparent functions. Every $f \in$ PRIMFN must satisfy the following requirements:

- f returns $\perp$ when invoked with a reference argument, or a variable that contains a member of REF.
- f neither reads nor updates unbound variables—the freelist stream included.
- f calls no other functions, except possibly itself.
- f returns a member of ATOM.

Successive calls to *freelist*( ) return successive elements from the freelist. Calling *freelist*( ) when there are no more elements in the stream of values causes a program to fail.

Most language constructs have their usual meaning. The statement "**assert** *cond*" is shorthand for "**if not** *cond* **then fail fi**". The *case* statement is equivalent to a nested if-then-else statement. A case statement causes a program to fail when none of its guards are matched.

7.3.2. Reducing pointer-language programs to pointer-free programs

A computation in language $\mathcal{H}$ is reduced to a computation in language $\mathcal{S}$ by using sets of special variables and values to simulate the heap; operations on variables to simulate expression evaluation; and combinations of operations to simulate statement evaluation.

Stores. A pointer store $\sigma_{\mathcal{H}}$ is reduced to a language $\mathcal{S}$-store by mapping every accessible structure in $\sigma_{\mathcal{H}}$ to a set of specially-named variables.[11] Let $\sigma_{\mathcal{H}}$, for example, be a store that contains n accessible structures. Then $\sigma_{\mathcal{H}}$ is reduced to a set of variables $\{ \mathrm{STR}_0.\mathit{suffix}_0, \cdots, \mathrm{STR}_0.\mathit{suffix}_{k-1}, \mathrm{STR}_1.\mathit{suffix}_0, \cdots, \mathrm{STR}_0.\mathit{suffix}_{k-1}, \cdots \mathrm{STR}_{n-1}.\mathit{suffix}_0, \cdots, \mathrm{STR}_{n-1}.\mathit{suffix}_{k-1} \}$, where each set of variables $\{ \mathrm{STR}_i.\mathit{suffix}_0, \cdots, \mathrm{STR}_i.\mathit{suffix}_{k-1} \}$ simulates an accessible structure in $\sigma_{\mathcal{H}}$, and each variable $\mathrm{STR}_i.\mathit{suffix}_j$ characterizes some aspect of the structure that it simulates. The suffix_j are strings that vary according to the computation being simulated. Assume, for example, that a program P manipulates one identifier, x; a procedure with two formal parameters, x and y; and a set of declared structures with one selector, INTP. Assume, furthermore, that $\sigma_{\mathcal{H}}$ only contains references of type x, y, and INTP. Then the reduction maps every structure s in $\sigma_{\mathcal{H}}$ to the following ten variables:

- STR_i._TYPE, which gives the type of s.
- STR_i._CURR, which references the current local environment when s is the global environment.
- STR_i._PREV, which references the previous local environment when s is an environment.
- STR_i._δ_1, which references a call site's first actual parameter when s is a local environment.

[11] A structure s in a store $\sigma_{\mathcal{H}}$ is *accessible* if there a path in $\sigma_{\mathcal{H}}$ from $\sigma_{\mathcal{H}}$'s global environment to s.

- STR_i._δ_2, which references a call site's second actual parameter when s is a local environment.
- STR_i._ATOM, which contains the atomic value of s if s is an atom.
- STR_i._TMP, which represents the value of field _TMP in s if s is the global environment.
- STR_i.X, which represents the value of field x in s if s is an environment.
- STR_i.Y, which represents the value of field y in s if s is a local environment.
- STR_i.INTP, which represents the value of field *intp* in s if s is of type "integer pointer".

In general, the number of $suffix_j$'s is five (_TYPE, _CURR, _PREV, _ATOM, _TMP), plus the maximum number of parameters for any one procedure, plus the number of distinct identifiers and selectors that are (1) mentioned in a program's text and (2) present in the initial store.

Structures. An atomic structure s in a pointer store σ is simulated as follows. Let s be the ith structure in σ, under some indexing scheme. Assume that s's value is a. Then s is simulated by a set of variables of the form STR_i.*suffix*, where STR_i._TYPE contains **atom**, and STR_i._ATOM contains a.

Structures that contain references are simulated using values from REF. A value **&i** is treated as a simulated reference to the structure STR_i. Assume, for example, that the local environment *lenv* is the jth structure in store $\sigma_{\mathcal{H}}$. Assume, furthermore, that field x in *lenv* references the ith structure in $\sigma_{\mathcal{H}}$. Then s is simulated by a set of variables of the form STR_j.*suffix*, where STR_j._TYPE contains **env**, and STR_j.X contains **&i**.

Variables that simulate unused fields in structures and variables in STR_i that simulate structures in a program's freelist are initialized to $\perp$. Variable set STR_0 is reserved for the global environment; the precise strategy for assigning indices to other simulated structures is left specified.

Expression evaluation. Language $\mathcal{H}$ supports four kinds of expressions: expressions that involve the relational operators **Eq**, <, =, and >; expressions that involve logical operators **typeOf** and **not**; store access expressions; and expressions that allocate storage. Expressions in $\mathcal{H}$ that involve relational and logical operators can be simulated directly in $\mathcal{S}$. Slightly more elaborate techniques are needed to simulate dereferencing and allocation.

Dereferencing. The evaluation of a store-access expression is simulated with case and assert statements. Figure 7.5 illustrates the simulated evaluation of an example identifier expression, *x.hd.tl*, in the context of the example assignment statement "[p] *x.hd.tl* := 10". The outer case statement simulates the application of the x selector to STR_0, the simulated global environment. The inner case statements simulate the application of *hd* to the structure denoted by x. The assert statements shown in this figure perform simulated run-time type checks; the meaning of STR_b.hd, for example, is undefined when STR_b fails to simulate an object of type *cons*.

The value **&maxref** (*cf.* Figure 7.5) is the largest simulated reference that a given reduction supports. This prespecified limit ensures that reduced programs are finite. The existence of **&maxref** also limits the accuracy of the reduction. Simulated computations, for example, that allocate more than **&maxref** locations must fail, due to storage overflow.

The technique for simulating "[p] *x.hd.tl* := 10" illustrated in Figure 7.5 assumes that x is *not* a local identifier at statement [p]. If x had been a local identifier, then the evaluation of *x.hd.tl* would have been simulated by making the implicit dereferencing of the pointer to the local environment explicit; *i.e.*, by first converting [p] to the equivalent statement "[p'] *_curr.x.hd.tl* := 10", and then using three

```
case STR0.X in
  &0: ···
  ···
  &b: assert STRb.TYPE is cons;
      case STRb._HD in
        &0: ···
        ···
        &c: assert STRc.TYPE is cons;
            STRc.TL := 10
        ···
        &maxref: ···
      esac;
  ···
  &maxref: ···
esac;
```

Figure 7.5. The simulation of "[p] *x.hd.tl* := 10", subject to the assumption that x is a global identifier at p. **&maxref** is a prespecified limit on the range of simulated references.

levels of case statements to simulate the evaluation of *_curr*, *x*, and *hd*.

Allocation. The evaluation of the expression "**new**(*conscell*)" is simulated in the manner depicted in Figure 7.6. The call to **freelist** obtains a reference to an unused STR_i from the simulated freelist. Three allocations are performed because fields of newly allocated structures are initialized to reference nil-valued atoms. The *freelist* operator also simulates the evaluation of atoms and primitive functions, which add new atoms to the store.

Throughout this chapter, it is assumed that every freelist passed to $\mathbf{M}_S$ is properly initialized. All values in the freelist must be between **&0** and **&maxref**. Furthermore, the value **&x** must not be present in the initial freelist if STR_x simulates an accessible structure in the initial store.

Executable statements. The subset of language $\mathcal{H}$ that is to be reduces to language S supports three kinds of statements: assignments; conditionals; and initialization statements. Figure 7.5 illustrates the simulation of an assignment statement. The simulation of conditionals and initialization statements is straightforward.

This completes the description of the reduction. It can easily be seen that the expressive power of the reduction is limited. The *a priori* bound on the domain of references precludes the simulation of stores that contain arbitrarily many structures. The use of **&maxref** also precludes the use of an arbitrarily long simulated freelist. Finally, the lack of calls and loops in S precludes the simulation of non-terminating programs.

The proof developed below copes with these limitations by reducing *loop-free programs* in language $\mathcal{H}$ w.r.t. *finite stores*. If a language $\mathcal{H}$ program P is known to terminate w.r.t. a finite store σ, then the number of structures that P allocates w.r.t. σ can be counted, and $P : \sigma$'s pattern of evaluation recorded. The characterization of $P : \sigma$ thereby obtained allows the creation of a reduced P that correctly evaluates a reduced

```
case freelist( ) in
  &0:  ···
  ···
  &a:  STR_a.TYPE := cons;
       case freelist( ) in          /*** *** initialize HD field of new structure to reference nil atom
         &0:  ···
         ···
         &b:  STR_b.TYPE := atom; STR_b.ATOM := nil; STR_a.HD := &b;
         ···
         &maxref:  ···
       esac;
       case freelist( ) in          /*** *** initialize TL field of new structure to reference nil atom
         &0:  ···
         ···
         &c:  STR_c.TYPE := atom; STR_c.ATOM := nil; STR_a.TL := &c;
         ···
         &maxref:  ···
       esac;
  ···
  &maxref:  ···
esac;
```

Figure 7.6. Simulating an occurrence of the expression "**new** (*conscell*)".

σ. This approach to reasoning about pointer programs yields assertions about the relationship between a program's *hsdg* and its *terminating* executions. (Theorems that specify how *hsdgs* characterize non-terminating programs are beyond the scope of this thesis.)

7.3.3. An equivalence lemma for language $\mathcal{S}$

The starting point for the proof of the Pointer-Language Equivalence Theorem is an equivalence theorem for a type of *dbr* referred to here as the *spdg*—*i.e.*, the language $\mathcal{S}$ *pdg*. A formal definition of the *spdg* is given in Appendix 8. Informally, the *spdg* is similar to the Horwitz-Prins-Reps (HPR) *pdg* described in Section 7.1: both *dbrs* portray a program's control, flow, and def-order dependences. There are also two important differences between the two *dbrs*. The first is that the language supported by the *spdg* (as defined here) does not support loops. There is therefore no need to distinguish between loop-carried and loop-independent dependences in *spdgs*. The *spdg* also portrays a more dynamic notion of data dependence: *i.e.*, one that accounts for a program's possible executions. For example, an HPR *pdg* for the program

"[1] **if** *pred* **then** [2] $x := 1$ **fi**; [3] $y := x$"

must contain the edge [2] $\longrightarrow_f$ [3]—even if *pred* is uniformly false. An *spdg* for P, on the other hand, may omit the edge [2] $\longrightarrow_f$ [3] when (*e.g.*) statement [2] never evaluates. A language $\mathcal{S}$ program may therefore have more than one valid *spdg*; this observation is similar to the observation that a language $\mathcal{H}$ program may have more than one *hsdg*.

The *Spdg Equivalence Theorem*, whose statement is given below, asserts that two programs that have isomorphic *spdgs* w.r.t. a set of inputs *InSet* also have equivalent behaviors w.r.t. *InSet*.

DEFINITION (***value computed at a program point (language S)***). The value computed at an assignment statement p is the value that p assigns to the variable on the left-hand-side of the assignment statement. The value computed at a predicate is the (boolean) value of the predicate. □

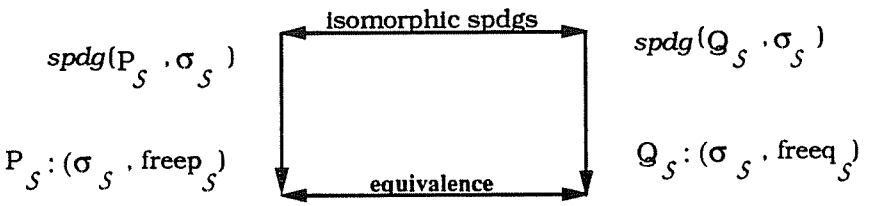
DEFINITION (***equivalent computations (language S)***). Let P_S and Q_S be programs in language S such that there exists an isomorphism f between the points of P_S and Q_S. Let $input_P$ and $input_Q$ be inputs. Let c_P denote the computation $P_S : input_P$, and c_Q the computation $Q_S : input_Q$. c_P and c_Q are *equivalent* w.r.t. f, written $c_P \approx_S c_P$ (w.r.t. f), iff

(1) neither c_P nor c_Q terminates successfully, or
(2a) c_P and c_Q both terminate successfully;
(2b) c_P and c_Q generate identical sequences of values at corresponding program points (w.r.t. f); and
(2c) the final stores computed by c_P and c_Q agree on the final values of all variables. □

LEMMA (***Spdg Equivalence Theorem***). Let $P \in Program_S$ and $Q \in Program_S$ be programs. Let f be an isomorphism between these programs' points. Let *InSet* be a set of inputs. Let G_P be an *spdg* for P w.r.t. *InSet*. Let G_Q be an *spdg* for Q w.r.t. *InSet*. Assume that G_P and G_Q are isomorphic w.r.t. f. Let $input_p = (\sigma, freep) \in InSet$ and $input_q = (\sigma, freeq) \in InSet$ be inputs such that $freep = freeq$. Then $P : input_p$ and $Q : input_q$ are equivalent computations. ∇

COROLLARY. The theorem also holds when $free \neq free'$ and neither P nor Q invokes **freelist**(). ∇

The pictorial characterization of the *Spdg* Equivalence Theorem, which was given in the preface to Section 7.3, is repeated here for convenience.



The *Spdg* Equivalence Theorem is proved by using a graph-rewriting semantics for *pdgs* to compare the evaluation of P and Q. A sketch of a proof for a related theorem is given in [Pfe91a]. The theorem proved there, which is due to Selke, concerns an extended *spdg* for an enhanced S that supports loops.

The lemma developed in this section, the *Simulation Equivalence Lemma*, uses the *Spdg* Equivalence Theorem to show that programs with related *spdgs* exhibit equivalent behavior when run on related inputs. The statement and proof of this theorem now follow.

DEFINITION (***flattened program (language S)***). Let P_S be a program in language S. Program P_S is *flattened* if every conditional statement in P_S is of one of two forms:

* "**if** *pred* **then fail else** ··· **fi**"
* "**if** *pred* **then** ··· **else fail fi**" □

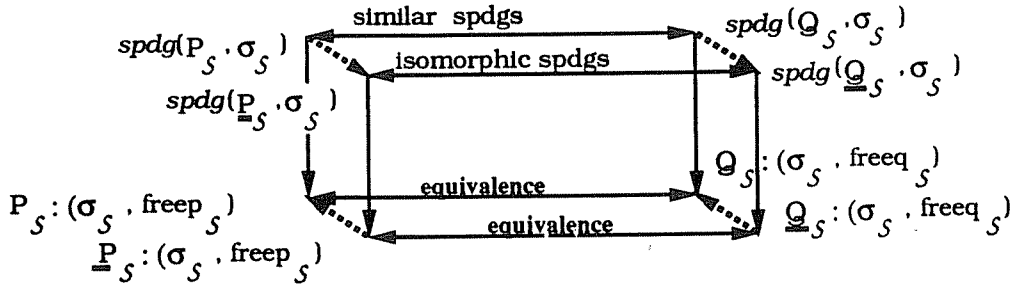
LEMMA (***Simulation Equivalence Lemma***). Let P_S and Q_S be flattened programs in language S such that there exists an isomorphism f between the points in P_S and Q_S. Assume that P_S and Q_S contain n

occurrences of the **freelist()** operator. Let $input_P = (\sigma_S, freep_S)$, where $freep_S$ is an n-element freelist that names none of the accessible STR_i in σ_S. Let $input_Q = (\sigma_S, freeq_S)$, where $freeq_S$ is that permutation of $freep_S$ whose jth element is determined as follows:

- Let q be the jth program point in Q_S that contains the **freelist()** operator (*N.B.*: this ordering is well-defined for flattened programs);
- Let p be the point that corresponds to q under f;
- Then the jth element of $freeq_S$ is the kth element of $freep_S$, where p is the kth program point in P_S that contains the **freelist()** operator.

Let G_P be an *spdg* for P_S w.r.t. $input_P$, and G_Q an *spdg* for Q_S w.r.t. $input_Q$. Assume that G_P and G_Q are similar *spdgs* w.r.t. f (*i.e.*, isomorphic up to freelist-mediated dependences), and that $P_S : input_P$ terminates successfully w.r.t. $input_P$. Then $P_S : input_P$ and $Q_S : input_Q$ are equivalent computations.

PROOF. The Simulation Equivalence Lemma is proved by reducing it to the *Spdg* Equivalence Theorem.



The first step in the proof reduces P_S and Q_S to comparable programs in a subset of S that lacks **freelist()** operators. Let $\underline{P}_S$ be the program obtained from P_S by replacing freelist operators in P_S with reference constants. In particular, the **freelist()** operator at the jth point in P_S is replaced with the expression **&c**, where **&c** is the jth reference in $freep_S$. Similarly, let $\underline{Q}_S$ be the program obtained by using $freeq_S$ to replace **freelist()** operators in Q_S. The assumptions made about the relative order of $freep_S$ and $freeq_S$ ensure that $\underline{P}_S$ and $\underline{Q}_S$ have isomorphic sets of points.

The second step of the proof establishes the relationship between the untransformed and transformed programs. Clearly, (•1) $P_S : input_P$ and $\underline{P}_S : input_P$ are identical computations, up to the state of their freelists. The hypothesis that P_S is a flattened program fixes the order in which values may be read from P_S's simulated freelist. Furthermore, the assumption that $P_S : input_P$ terminates successfully implies that $P_S : input_P$ must not exhaust its simulated freelist. Computations $Q_S : input_Q$ and $\underline{Q}_S : input_Q$ are also identical computations, up to the state of their freelists. This assertion, however, is established indirectly, by first comparing $\underline{P}_S : input_P$ to $\underline{Q}_S : input_Q$.

The next step of the proof shows that (•2) $\underline{P}_S : input_P$ and $\underline{Q}_S : input_Q$ are equivalent computations. This assertion is demonstrated by constructing isomorphic *spdgs* for $\underline{P}_S$ and $\underline{Q}_S$. Assertion (•2) then follows from the corollary to the *Spdg* Equivalence Theorem, since neither $\underline{P}_S$ nor $\underline{Q}_S$ access their freelists.

Let $\underline{G}_P$ be the *dbr* obtained from G_P by (1) relabeling every vertex in G_P that corresponds to an operation on the freelist with the corresponding operation (on reference constants) in $\underline{G}_P$, and (2) removing all edges from the resulting graph that represent freelist-mediated dependences. Let $\underline{G}_Q$ be obtained

from G_Q in a similar manner. Clearly, $\underline{G_P}$ and $\underline{G_Q}$ are isomorphic *dbrs*. It must now be shown that $\underline{G_P}$ and $\underline{G_Q}$ are valid *spdgs* for $\underline{P_S}$ and $\underline{Q_S}$, relative to their respective inputs. Assume, on the contrary, that (*e.g.*) $\underline{G_P}$ fails to represent a (dynamic) data dependence $p \rightarrow_d q$ exhibited by $\underline{P_S}:input_P$. Then, by the definition of data dependence, $\underline{P_S}:input_P$ must contain a sequence of states that gives rise to $p \rightarrow_d q$. The close relationship between P_S and $\underline{P_S}$, however, now ensures that a comparable sequence of states gives rise to a comparable, *missing* dependence in G_P—a contradiction. A similar argument shows that $\underline{G_Q}$ is an *spdg* for $\underline{Q_S}$ w.r.t. $input_P$.

This completes the demonstration of assertion (•2). Assertions (•1) and (•2) are now used to demonstrate that (•3) $Q_S:input_Q$ and $\underline{Q_S}:input_Q$ are identical computations, up to the state of their freelists. Note, first of all, that assertion (•1) and the successful termination of $P_S:input_P$ imply that (*) $\underline{P_S}:input_P$ terminates successfully. Furthermore, assertion (*) and assertion (•2) now imply that $\underline{Q_S}:input_Q$ terminates successfully—*i.e.*, that the evaluation of $\underline{Q_S}:input_Q$ followed the lone non-failing path through $\underline{Q_S}$. An induction on the number of statements in $\underline{Q_S}$ and Q_S now completes the proof of assertion (•3).

Summarizing the argument so far, there exist two programs $\underline{P_S}$ and $\underline{Q_S}$ that satisfy the following three assertions:

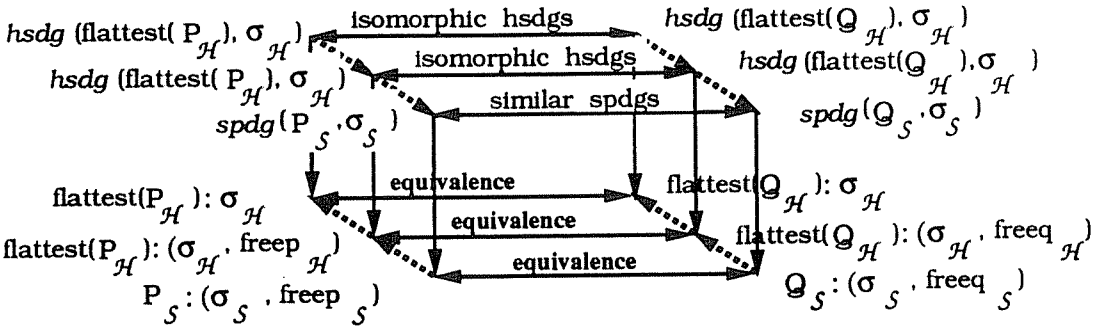
- •1. Computations P_S:$input_P$ and $\underline{P_S}$:$input_P$ are identical, up to the state of their freelists.
- •2. Computations $\underline{P_S}$:$input_P$ and $\underline{Q_S}$:$input_Q$ compute identical sequences of values at corresponding program points, and compute the same final stores.
- •3. Computations $\underline{Q_S}$:$input_Q$ and Q_S:$input_Q$ are identical, up to the state of their freelists.

Taken together, these assertions imply that P_S:$input_P$ and Q_S:$input_Q$ are equivalent computations. □

A final observation is in order about $freep_S$ and $freeq_S$. The decision to assume that $freeq_S$ was a specific permutation of $freep_S$, in effect, represented a decision to handle all reasoning about referential transparency at the level of language $\mathcal{H}$ (*cf.* proof step 5 in the preface to §7.3). An earlier draft of this thesis attempted to develop a version of the Simulation Equivalence Lemma that characterized the effect of evaluating P_S and Q_S w.r.t. *unrelated* freelists. This approach, although reasonable in principle, proved quite unattractive in practice: the fact that a language $\mathcal{S}$ computation is sensitive to the specific values of reference constants greatly complicates the statement—and proof—of this alternative theorem.

7.3.4. Using the reduction to map from $\mathcal{H}$ to $\mathcal{S}$

The current section presents the fifth and sixth steps in the proof of the Pointer-Language Equivalence Theorem: *i.e.*, the steps that reduce a pair of flattened language $\mathcal{H}$ programs to a comparable pair of language $\mathcal{S}$ programs.



The principal assertion proved in this section, the *Flattened Programs Equivalence Theorem*, states that flattened programs with isomorphic *hsdgs* have equivalent behavior. The Flattened Programs Equivalence Theorem is proved in two stages. The first stage demonstrates the correctness of the reduction described in the sixth step of the proof sketch (*i.e.*, the diagram's "foreground brick"). This first result is termed the *Flattened Programs Lemma*. The Flattened Programs Lemma is then used to demonstrate the correctness of the reduction described in the fifth step of the sketch (*i.e.*, the diagram's "background brick").

The Flattened Programs Lemma is proved by using the translation described in Section 7.3.2 to reduce computations in language $\mathcal{H}$ to comparable computations in language $\mathcal{S}$. The statement of this lemma is formalized with the aid of the definitions given below. The first four definitions establish terminology for comparing computations in language $\mathcal{H}$ and $\mathcal{S}$. The next two definitions formalize the reduction. The final three definitions formalize the notion of equivalent computations in language $\mathcal{H}$.

DEFINITION (***congruent program points***). Let $pt_{\mathcal{H}}$ be a point in a language $\mathcal{H}$ program $P_{\mathcal{H}}$, and $pt_{\mathcal{S}}$ a point in a reduced version of this program, program $P_{\mathcal{S}}$. Points $pt_{\mathcal{H}}$ and $pt_{\mathcal{S}}$ are *congruent* (w.r.t. $P_{\mathcal{H}}$ and $P_{\mathcal{S}}$) iff $pt_{\mathcal{S}}$ is the first of the points in $P_{\mathcal{S}}$ that simulates $pt_{\mathcal{H}}$. □

DEFINITION (***congruent stores***). Let $\sigma_{\mathcal{H}}$ be a member of $Store_{\mathcal{H}}$. Let $\sigma_{\mathcal{S}}$ be a member of $Store_{\mathcal{S}}$. Let *Idexp* be the set of *all* path expressions in language $\mathcal{H}$. Stores $\sigma_{\mathcal{H}}$ and $\sigma_{\mathcal{S}}$ are *congruent* iff

- for all *idexp* ∈ *Idexp*, *idexp* denotes a structure of type t w.r.t. $\sigma_{\mathcal{H}}$ iff *idexp* denotes a STR_i w.r.t. $\sigma_{\mathcal{S}}$ with STR_i.TYPE = t;
- for all *idexp* ∈ *Idexp*, *idexp* denotes the atom v in $\sigma_{\mathcal{H}}$ iff *idexp* denotes a STR_i w.r.t. $\sigma_{\mathcal{S}}$ with STR_i.ATOM = v and STR_i.TYPE = **atom**; and
- for all *idexp* ∈ *Idexp* and *idexp'* ∈ *Idexp*, *idexp* and *idexp'* are aliases w.r.t. $\sigma_{\mathcal{H}}$ iff *idexp* and *idexp'* are aliases w.r.t. $\sigma_{\mathcal{S}}$. □

This definition of congruence is an isomorphism between the accessible structures in $\sigma_{\mathcal{H}}$ and the accessible STR_i in $\sigma_{\mathcal{S}}$.

DEFINITION (***congruent states***). Two states are *congruent states* if their program-point and store components are congruent. □

DEFINITION (***congruent computations***). Let $c_{\mathcal{H}} = (h_1, h_2, \cdots)$ be a computation in language $\mathcal{H}$, and $c_{\mathcal{S}} = (s_1, s_2, \cdots)$ a computation in language $\mathcal{S}$. Computations $c_{\mathcal{H}}$ and $c_{\mathcal{S}}$ are ***congruent computations***, written $c_{\mathcal{H}} \approx_{\mathcal{HS}} c_{\mathcal{S}}$, if there is a map f from $c_{\mathcal{H}}$ to $c_{\mathcal{S}}$ such that

- f maps every state h_i to a congruent state s_j; and
- if $f(h_i) = s_j$ and $f(h_{i+1}) = s_k$, then $j < k$, and states $s_j \cdots s_{k-1}$ simulate the evaluation of the point at h_i. □

DEFINITION (***reduceStore***). The function $reduceStore : Int \times Program_{\mathcal{H}} \times Store_{\mathcal{H}} \rightarrow Store_{\mathcal{S}}$ reduces stores in language $\mathcal{H}$ to congruent stores in language $\mathcal{S}$, according to the rules described in Section 7.3.2. In particular, let *maxref* be *reduceStore*'s first parameter, and *used* denote the number of accessible structures in *reduceStore*'s third parameter; then STR_0 through STR_{used-1} simulate accessible structures in the store, and STR_{used} through STR_{maxref} simulate structures in the initial freelist. The second and third parameters determine the variables that make up the STR_i's. □

DEFINITION (***reducePgm***). The function $reducePgm : Int \times Program_{\mathcal{H}} \rightarrow Program_{\mathcal{S}}$ reduces a program $P_{\mathcal{H}}$ in language $\mathcal{H}$ to a program in language $\mathcal{S}$. This reduction is accomplished by using the rules given in Section 7.3.2 to reduce the statement list obtained by in-line expanding the call to procedure *main*() at point $\mathbf{initial}_2$. (*N.B.*: An in-line-expansion of a procedure call is illustrated in Figure 7.7.) *reducePgm*'s first parameter is (the integer value of) **&maxref**. □

Intuitively, the *reducePgm*() function performs exactly one in-line expansion because a flattened program performs exactly one procedure call. This procedure call, an implicit call to procedure *main*(), immediately follows the initialization of a program's store (*cf.* Chapter 2). This call to *main*() at point $\mathbf{initial}_2$ must be eliminated before a program can be translated into language $\mathcal{S}$.

DEFINITION (***value computed at a program point (language $\mathcal{H}$)***). The value computed at an assignment statement p is the value that p assigns to the variable on the left-hand-side of the assignment statement. The value computed at a predicate is the (boolean) value of the predicate. □

DEFINITION (***equivalent values, sequences of values (language $\mathcal{H}$)***). Two values v_1 and v_2 are *equivalent* iff either v_1 and v_2 are atoms and $v_1 = v_2$, or v_1 and v_2 are both references.

```
_curr._δ1 := a1                          /*** initialize actual parameters
...
_curr._δk := ak
_tmp := _curr ;                          /*** initialize new environment
_curr := new(env);
_curr._prev := _tmp ;
_curr._f1 := _curr._prev._δ1             /*** initialize formal parameters
...
_curr._fk := _curr._prev._δk
body_A ;                                 /*** evaluate procedure
_curr := _curr._prev ;                   /*** restore old environment
```

Figure 7.7. An in-line-expanded procedure call. Procedure A has k formals $f_1 \cdots f_k$ and body $body_A$. The expanded call statement is "*call* $A(a_1, \cdots, a_k)$.

Let $v = v_1 \cdots v_m$ and $w = w_1 \cdots w_n$ be sequences of values. Sequences v and w are *equivalent* iff $m = n$ and v_i and w_i are equivalent for all i between 1 and m inclusive. □

DEFINITION (***equivalent computations (language $\mathcal{H}$)***). Let $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ be programs in language $\mathcal{H}$ such that there exists an isomorphism f between the points of $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. Let $\sigma_{\mathcal{H}}$ be a store. Let c_P denote the computation $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$, and c_Q the computation $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$. c_P and c_Q are *equivalent* w.r.t. f, written $c_P \approx_{\mathcal{H}} c_P$ (w.r.t. f), iff either

(1) neither c_P nor c_Q terminates successfully, or
(2a) c_P and c_Q both terminate successfully;
(2b) c_P and c_Q compute equivalent sequences of values at corresponding points (w.r.t. f); and
(2c) the final stores computed by c_P and c_Q are equivalent. □

The statement and proof of the Flattened Programs Lemma now follow.

DEFINITION The expression $P_{\mathcal{H}}: (\sigma_{\mathcal{H}}, fl_{\mathcal{H}})$ denotes the computation of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ w.r.t. a *specific* $fl_{\mathcal{H}} \in Free$. □

LEMMA (***Flattened Programs Lemma***). Let $\sigma_{\mathcal{H}}$ be a store, and $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ be flattened programs in language $\mathcal{H}$ that have k allocation sites.[12] Assume that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. $\sigma_{\mathcal{H}}$ and the two freelists $freep_{\mathcal{H}}$ and $freeq_{\mathcal{H}}$ defined below. Let f be this isomorphism between the points in $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$.

Let $freep_{\mathcal{H}}$ be a freelist that names k locations, none of which correspond to accessible locations in $\sigma_{\mathcal{H}}$. Let $freeq_{\mathcal{H}}$ be that permutation of $freep_{\mathcal{H}}$, $\pi(freep_{\mathcal{H}})$, whose jth element is determined as follows:

- Let q be the jth program point in $Q_{\mathcal{H}}$ that contains a **new**( ) operator (*N.B.*: this ordering is well-defined for flattened programs);
- Let p be the point that corresponds to q under f;
- Then the jth element of $freeq_{\mathcal{H}}$ is the kth element of $freep_{\mathcal{H}}$, where p is the kth program point in $P_{\mathcal{H}}$ that contains a **new**( ) operator.

Let $inputp_{\mathcal{H}} = (\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ and $inputq_{\mathcal{H}} = (\sigma_{\mathcal{H}}, freeq_{\mathcal{H}})$. If $P_{\mathcal{H}}: inputp_{\mathcal{H}}$ terminates successfully, then $P_{\mathcal{H}}: inputq_{\mathcal{H}} \approx_{\mathcal{H}} Q_{\mathcal{H}}: inputq_{\mathcal{H}}$; *i.e.*, the two computations are equivalent.

PROOF (*sketch*). This lemma is proved by reducing it to the Simulation Equivalence Lemma. Let:

n denote the number of accessible structures in $\sigma_{\mathcal{H}}$;
$\sigma_S = reduceStore(n+k-1, P_{\mathcal{H}}, \sigma_{\mathcal{H}})$;
$P_S = reducePgm(n+k-1, P_{\mathcal{H}})$; $\quad Q_S = reducePgm(n+k-1, Q_{\mathcal{H}})$;
$freep_S = \&\mathbf{n} \cdots \&\mathbf{n+k-1}$; $\quad freeq_S = \pi(freep_S)$
$inputp_S = (\sigma_S, freep_S)$; and $\quad inputq_S = (\sigma_S, freeq_S)$.

A straightforward case analysis of the type of a reduced statement can be used to show computations in $\mathcal{H}$ are mapped to congruent computations in S: *i.e.*, that $P_{\mathcal{H}}: inputp_{\mathcal{H}} \approx_{\mathcal{HS}} P_S: inputp_S$ and

[12]Technically, the implicit call to *main*( ) at the start of a program, which initializes *main*( )'s local environment, must also be counted as an allocation site.

$Q_{\mathcal{H}}$: $inputq_{\mathcal{H}} \approx_{\mathcal{HS}} Q_S$: $inputq_S$. The definitions of $\approx_{\mathcal{HS}}$, $\approx_S$, and $\approx_{\mathcal{H}}$ now imply that the equivalence of $P_{\mathcal{H}}$: $inputp_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $inputq_{\mathcal{H}}$ can be proved by demonstrating (*) the equivalence of P_S: $inputp_S$ and Q_S: $inputq_S$.

Assertion (*) will be shown by demonstrating that P_S, Q_S, $inputp_S$, and $inputq_S$ satisfy the hypotheses of the Simulation Equivalence Lemma (*cf.* §7.3.3). Specifically, it must be shown that (1) P_S: $inputp_S$ terminates; that (2) $freep_S$ and $freeq_S$ name k locations that are not accessible in σ_S; that (3) $freeq_S$ is a specified permutation of $freep_S$; and that (4) P_S and Q_S have *similar*[13] *spdgs*. Assertions (1) through (3), however, are almost immediate—(2) and (3) follow from the construction of $freep_S$ and $freeq_S$, and (1) from the congruence of $P_{\mathcal{H}}$: $inputp_{\mathcal{H}}$ and P_S: $inputp_S$. Assertion (4)—the assertion that P_S and Q_S have similar *spdgs*—is demonstrated below.

A hypothesis of the Flattened Programs Lemma states that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs*. Let $GP_{\mathcal{H}}$ and $GQ_{\mathcal{H}}$ be this pair of isomorphic *hsdgs*. $GP_{\mathcal{H}}$ and $GQ_{\mathcal{H}}$ can now used to construct isomorphic *dbrs* for P_S and Q_S, as follows. Let *sim* be a function that pairs every point p in P_S with the point in $P_{\mathcal{H}}$ that p simulates. Let

- *static* be the set of statically feasible (*i.e.*, control-flow-graph-derivable) data dependences exhibited by P;
- *freedep* be the set of freelist-mediated dependences in *static*;
- *notinH* be the set of $p \rightarrow_d q \in static$ such that $sim(p) \rightarrow_d sim(q)$ is not represented in $GP_{\mathcal{H}}$;
- $spurious = notinH - freedep$; and
- $datadep = static - spurious$.

Let GP_S be that *spdg* for P that depicts all of P's control dependences and every data dependence in *datadep*. Let GQ_S be constructed in a similar fashion. It can be verified that *dbrs* GP_S and GQ_S are similar *dbrs*. Assertion (4) can therefore be established by showing that (i) GP_S is a valid *spdg* for P_S w.r.t. $inputp_S$, and (ii) GQ_S is a valid *spdg* for Q_S w.r.t. $inputq_S$. Since the proofs of (i) and (ii) are similar, only the first is given.

Since GP_S represents P_S's control dependences, assertion (i) can be proven by showing that GP_S represents all of P_S's dynamic data dependences w.r.t. $inputp_S$. Assume, on the contrary, that GP_S fails to depict a $d_S = p \rightarrow_d q$ exhibited by P_S: $inputp_S$. Then the definition of GP_S implies that $d_S \in spurious$; *i.e.*, that $sim(p) \rightarrow_d sim(q)$ is a non-freelist-mediated dependence that is absent from $GP_{\mathcal{H}}$. The definition of dependence w.r.t. S, however, also implies that P_S: $inputp_S$ contains a sequence of states that exhibits d_S. Furthermore, the congruence of P_S: $inputp_S$ and $P_{\mathcal{H}}$: $inputp_{\mathcal{H}}$ implies that the corresponding sequence of states in $P_{\mathcal{H}}$: $inputp_{\mathcal{H}}$ gives rise to $sim(p) \rightarrow_d sim(q)$. This observation that $P_{\mathcal{H}}$: $inputp_{\mathcal{H}}$ exhibits $sim(p) \rightarrow_d sim(q)$ contradicts the hypothesis that $GP_{\mathcal{H}}$ is a valid *hsdg* for $P_{\mathcal{H}}$ w.r.t. $inputp_{\mathcal{H}}$. GP_S must therefore be a valid *spdg* for program P_S. □

The statement and proof of the Flattened Programs Equivalence Theorem now follows.

---

[13] Recall that two *spdgs* are similar iff they are isomorphic, up to edges that represent freelist-mediated dependences.

LEMMA (***Flattened Programs Equivalence Theorem***). Let $\sigma_{\mathcal{H}}$ be a store. Let $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ be flattened programs in language $\mathcal{H}$ such that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. $\sigma_{\mathcal{H}}$. Assume, furthermore, that $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ terminates successfully. Then $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent computations.

PROOF. Recall that the definition of $\mathcal{H}$ leaves the freelist component of a computation largely unspecified. The expression $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$, in effect, denotes the evaluation of $P_{\mathcal{H}}$ w.r.t. $\sigma_{\mathcal{H}}$ and *any* infinite freelist $fl_{\mathcal{H}}$, where every location in $fl_{\mathcal{H}}$ is inaccessible in $\sigma_{\mathcal{H}}$. The freelist component is not specified because fixing the contents of the list does not affect a pointer program's behavior in any material way. More precisely, assume that $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ allocates no more than k locations. Let $Free_k$ denote the set of all freelists that (1) contain at least k locations such that (2) none of these locations correspond to accessible structures in $\sigma_{\mathcal{H}}$. Let $freep_{\mathcal{H}}$ and $freeq_{\mathcal{H}}$ be arbitrary elements of $Free_k$. A straightforward induction shows that $P_{\mathcal{H}}: (\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ and $P_{\mathcal{H}}: (\sigma_{\mathcal{H}}, freeq_{\mathcal{H}})$ generate sequences of states that have identical program-point components and isomorphic store components. This observation also implies that $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ exhibits the same data dependences w.r.t. any initial freelist in $Free_k$. [14]

Let k be the number of allocation sites in $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. Since $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are straight-line programs, any computation involving $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ allocates no more than k structures. Let $Free_k$ be defined as in the previous paragraph. The observations in the previous paragraph, together with the assumption that $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are loop-free programs, implies that any $freep_{\mathcal{H}}$ and $freeq_{\mathcal{H}}$ in $Free_k$ can be used to reason about $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$, respectively. Furthermore, the observations about a computation's dependences w.r.t. a specific freelist imply that G_P is an *hsdg* for $P_{\mathcal{H}}$ w.r.t. $\sigma_{\mathcal{H}}$ iff G_P is also an *hsdg* w.r.t. $\sigma_{\mathcal{H}}$ and $freep_{\mathcal{H}}$.

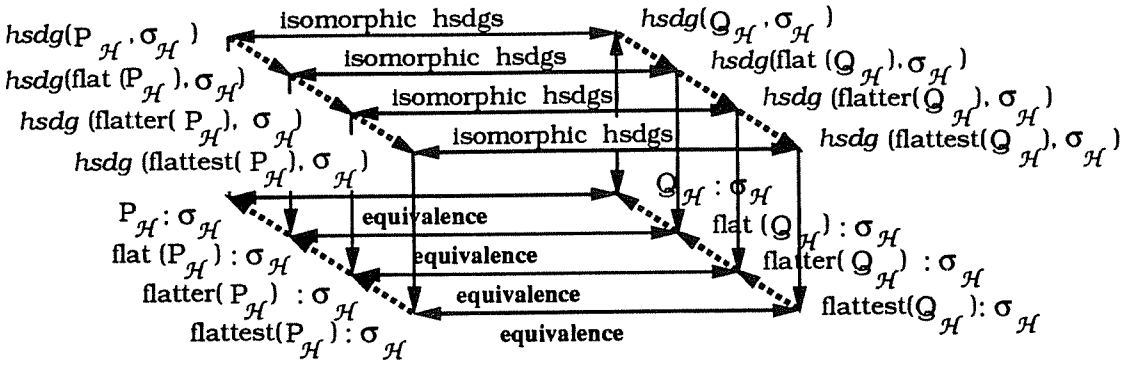
To summarize the argument to this point, $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent computations iff there exist two freelists in $Free_k$, $freep_{\mathcal{H}}$ and $freeq_{\mathcal{H}}$, such that $P_{\mathcal{H}}: (\sigma_{\mathcal{H}}, freep_{\mathcal{H}})$ and $Q_{\mathcal{H}}: (\sigma_{\mathcal{H}}, freeq_{\mathcal{H}})$ are equivalent computations. Let $freep_{\mathcal{H}}$ be an arbitrary member of $Free_k$, and $freeq_{\mathcal{H}}$ be that permutation of $freep_{\mathcal{H}}$ that satisfies the hypotheses of the Flattened Programs Lemma. The observation that $P_{\mathcal{H}}$, $Q_{\mathcal{H}}$, G_P, G_Q, $\sigma_{\mathcal{H}}$, $freep_{\mathcal{H}}$, and $freeq_{\mathcal{H}}$ satisfy the hypotheses of the Flattened Programs Lemma now establishes the Flattened Programs Equivalence Lemma. □

7.3.5. Flattening programs in language $\mathcal{H}$

The current section discusses the second, third, and fourth steps in the proof of the Pointer-Language Equivalence Theorem: *i.e.*, the steps that reduce a pair of language $\mathcal{H}$ programs to a comparable pair of flattened language $\mathcal{H}$ programs.

---

[14] A similar result is demonstrated in Chapter 5; the proof that $evalPt_A$ is monotone uses the referential transparency of $\mathcal{H}$'s primitive operations to argue that replacing a store with an isomorphic store has no appreciable effect on a computation's outcome.



The three reductions described in this section use the assumption that $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ terminates to simplify $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. These reductions, which flatten conditionals, remove loops, and in-line expand procedure calls, will be presented in somewhat less detail than the transformations given in the preceding sections. This less formal style of presentation streamlines the proofs without detracting from the argument: formalizing the definitions of the various congruences would have complicated the proofs without really clarifying the intuition. A second reason for using this more relaxed style of presentation is that these types of reductions are not new. Previous authors have used flattening transformations to theorems about the *dbrs* of richer languages to theorems about simpler languages (*cf.* [Bin89, Sel90]). The only significant difference between the arguments developed here and these earlier proofs is that observations about a program's *execution* (rather than its control-flow graph) must be used to argue that reduced programs have isomorphic *hsdgs*. Furthermore, arguments about program behavior have already been used to establish comparable assertions about *dbrs* in Sections 7.3.3 and 7.3.4: this concern arises (*e.g.*) in the proof of the Flattened Programs Lemma, where it must be shown that the map from $\mathcal{H}$ to $\mathcal{S}$ reduces programs with isomorphic *hsdgs* to programs with similar *spdgs*.

Step 4. The fourth step in the proof of the Pointer-Language Equivalence Theorem assumes that

- $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are auxiliary-procedure-free, procedure-call-free, loop-free programs;
- $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. a store $\sigma_{\mathcal{H}}$; and
- $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ terminates successfully.

Step 4 then argues that (•4) $P_{\mathcal{H}}:\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}:\sigma_{\mathcal{H}}$ are equivalent computations. This is done by reducing $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ to comparable, flattened programs, and then applying the Flattened Programs Equivalence Theorem to complete the proof.

Define a conditional to be ***unflattened*** iff neither its true nor its false consequent is of the form **fail**. Let d be the maximum depth to which unflattened conditionals are nested in $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. Let n be the number of unflattened conditionals that enclose at least one unflattened conditional at depth d. Assertion (•4) is now shown with a double induction on d and n.

If $d = 0$, then $n = 0$, and $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are flattened programs. Assertion (•4) is then immediate from the Flattened Programs Equivalence Theorem (*cf.* previous section).

If $d > 0$ and $n > 0$, let pt_P be the program point of an unflattened conditional in $P_{\mathcal{H}}$ that (1) is not enclosed by any other unflattened conditionals, and (2) encloses an unflattened conditional at depth d. Let pt_Q be the corresponding point in $Q_{\mathcal{H}}$. Assume (*e.g.*) that pt_P evaluates to **true** during the execution of

$P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$. The false consequents of the conditionals at pt_P and pt_Q are then replaced with a single **fail** statement. This reduction decreases n and possibly d; it decreases n by 1, and decreases d by at least 1 when $n = 1$. This reduction also produces a pair of programs—call them $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$—that have isomorphic *hsdgs*. Furthermore, computations $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ are clearly congruent (*i.e.*, closely related) computations. The equivalence of $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ can now be established by establishing (i) the equivalence of $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and (ii) the congruence of $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$.

Assertion (i)—the equivalence of $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$—follows from the induction hypothesis and the isomorphism of $\underline{P}_{\mathcal{H}}$'s and $\underline{Q}_{\mathcal{H}}$'s *hsdgs*. Assertion (ii) is established using (i). In particular, (i) and the equivalence of $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ imply that (*) pt_Q evaluates to true in $\underline{Q}_{\mathcal{H}}$. Furthermore, (**) the same stores much reach pt_Q in $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$. Observations (*) and (**) now imply that pt_Q must evaluate to **true** in $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$—thereby establishing (ii) the congruence of $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$.

Step 3. The third step in the proof of the Pointer-Language Equivalence Theorem assumes that

- $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are auxiliary-procedure-free, procedure-call-free programs;
- $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. a store $\sigma_{\mathcal{H}}$; and
- $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ terminates successfully.

Step 3 then argues that (•3) $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ are equivalent computations. This is done by reducing $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ to related, loop-free programs, and then using step 4, assertion (•4) to complete the proof.

Let d be the maximum depth to which loops are nested in $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. Let n be the number of loops that enclose at least one loop at depth d. Assertion (•3) is now shown with a double induction on d and n.

If $d = 0$, then $n = 0$, and $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ are loop-free programs. Assertion (•3) is then immediate from assertion (•4).

If $d > 0$ and $n > 0$, let L_P be the program point of a loop in $P_{\mathcal{H}}$ that (1) is not enclosed by any other loops, and (2) encloses a loop at depth d. Let L_Q be the corresponding loop in $Q_{\mathcal{H}}$. The assumption that $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ terminates now fixes the number of times that the body of L_P evaluates w.r.t. $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ to some value, k. Let $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$ be the programs obtained by replacing L_P and L_Q with the k-ary approximation to these loops depicted in Figure 7.8. This reduction decreases n and possibly d; it decreases n by 1, and decreases d by at least 1 when $n = 1$. This reduction also produces a pair of programs—call them $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$—that have isomorphic *hsdgs*. (*N.B.*: the assertion that a data dependence is either carried by or independent of L_P (L_Q) must be used to guide the placement of edges in $\underline{P}_{\mathcal{H}}$'s ($\underline{Q}_{\mathcal{H}}$'s) *hsdg*.) Furthermore, computations $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ are clearly congruent computations. The equivalence of $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ can now be established by establishing (i) the equivalence of $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and (ii) the congruence of $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$.

Assertion (i)—the equivalence of $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$—follows from the induction hypothesis and the isomorphism of $\underline{P}_{\mathcal{H}}$'s and $\underline{Q}_{\mathcal{H}}$'s *hsdgs*. Assertion (ii) is established using (i). In particular, let $q_1 \cdots q_{k+1}$ be the $k + 1$ predicates in $\underline{Q}_{\mathcal{H}}$ that correspond, under the reduction, to L_Q's controlling predicate. Assertion (i) and the congruence of $P_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{P}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ imply that (*) $q_1 \cdots q_k$ evaluate to **true**, and q_{k+1} to **false**. An induction on k that uses the correspondence between $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ then establishes that (**) L_Q also evaluates k times in $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$. Assertion (**) implies (ii) the congruence of $\underline{Q}_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}$: $\sigma_{\mathcal{H}}$.

Example loop		k–ary approximation to loop	
[p]	**while** *pred* **do**	[$p.1$]	**if** *pred* **then**
	$body_1$		$body_1$;
	od	[$p.2$]	**if** *pred* **then**
			$body_1$;
		[$p.3$]	**if** *pred* **then**
			...
		[$p.k$]	**if** *pred* **then**
			$body_1$;
		[$p.k+1$]	**if** *pred* **then fail fi**
			fi*

Figure 7.8. A k-ary approximation to a while loop.

Step 2. The second step in the proof of the Pointer-Language Equivalence Theorem assumes that

- $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ have isomorphic *hsdgs* w.r.t. a store $\sigma_{\mathcal{H}}$; and
- $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ terminates successfully.

Step 2 then argues that (•**2**) $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent computations. This is done by reducing $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ to related, auxiliary-procedure-free, call-statement-free programs, and then using step 3, assertion (•**3**) to complete the proof.

Let c be the number of calls to auxiliary procedures performed during the evaluation of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$. Assertion (•**2**) is now shown with an induction on c.

When $c = 0$, the proof proceeds by an induction on the s, the number of call statements in the body of $P_{\mathcal{H}}$'s *main*( ) procedure.

If $s = 0$, then $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$'s *main*( ) procedures contain no calls on auxiliary procedures. Let $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$ be the programs obtained from $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ by removing their auxiliary procedures. Clearly, $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$ have isomorphic *hsdgs*. Assertion (•**3**) now implies that $\underline{P}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent computations. The observation that the unreduced and reduced computations are (trivially) congruent now establishes (•**2**).

If $s > 0$, let pt_P be a call site in $P_{\mathcal{H}}$, and pt_Q the corresponding call site in $Q_{\mathcal{H}}$. Since $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ makes no calls on auxiliary procedures, pt_P could not have evaluated during the execution of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$. Replacing pt_P with a **fail** statement therefore reduces $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ to a congruent computation. The argument is now completed by (1) replacing the corresponding call site in $Q_{\mathcal{H}}$ with a **fail** statement; (2) using the induction hypothesis to show the equivalence of the reduced computations, and (3) using the assumption that $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ succeeds to show that the replaced call site in $Q_{\mathcal{H}}$ could not have evaluated.

If $c > 0$, let pt_P be the program point of a call statement in $P_{\mathcal{H}}$'s *main*( ) procedure that was invoked during $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$. Let pt_Q be the corresponding call site in $Q_{\mathcal{H}}$. The calls at pt_P and pt_Q are now in-line-expanded, according to Figure 7.7. This reduction produces a pair of programs—call them $\underline{P}_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$—

that have isomorphic *hsdgs*. (*N.B.*: the assertion that a data dependence is either carried by or independent of pt_P (pt_Q) must be used to guide the placement of edges in $\underline{P}_{\mathcal{H}}$'s ($\underline{Q}_{\mathcal{H}}$'s) *hsdg*.) Furthermore, computations $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{P}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are clearly congruent computations. The equivalence of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ can now be established by establishing (i) the equivalence of $\underline{P}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and (ii) the congruence of $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$.

Assertion (i)—the equivalence of $\underline{P}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$—follows from the induction hypothesis and the isomorphism of $\underline{P}_{\mathcal{H}}$'s and $\underline{Q}_{\mathcal{H}}$'s *hsdgs*. Assertion (ii) is established using (i). In particular, (i) and the congruence of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{P}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ imply that (*) $\underline{Q}_{\mathcal{H}}$ evaluates the expanded procedure call at pt_Q. The similarity of $Q_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}$ also implies that (**) the same stores much reach any predicates that control the evaluation of pt_Q in $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$. Observations (*) and (**) now imply that call site pt_Q must evaluate in $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$—thereby establishing (ii) the congruence of $\underline{Q}_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$.

7.3.6. The Pointer-Language Equivalence Theorem

The statement and proof of the section's main theorem, the Pointer-Language Equivalence Theorem, are given below. This theorem, roughly speaking, states that programs with isomorphic *hsdgs* map equivalent inputs to equivalent final stores. The definition of the term *equivalent computations*, which was given in Section 7.3.4, is repeated here for convenience.

DEFINITION (***value computed at a program point (language $\mathcal{H}$)***). The value computed at an assignment statement p is the value that p assigns to the variable on the left-hand-side of the assignment statement. The value computed at a predicate is the (boolean) value of the predicate. □

DEFINITION (***equivalent values, sequences of values (language $\mathcal{H}$)***). Two values v_1 and v_2 are *equivalent* if either v_1 and v_2 are atoms and $v_1 = v_2$, or v_1 and v_2 are both references.
Let $v = v_1 \cdots v_m$ and $w = w_1 \cdots w_n$ be sequences of values. Sequences v and w are *equivalent* iff $m = n$ and v_i and w_i are equivalent for all i between 1 and m inclusive. □

DEFINITION (***equivalent computations (language $\mathcal{H}$)***). Let $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ be programs in language $\mathcal{H}$ such that there exists an isomorphism f between the points of $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$. Let $\sigma_{\mathcal{H}}$ be a store. Let c_P denote the computation $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$, and c_Q the computation $Q_S: \sigma_{\mathcal{H}}$. c_P and c_Q are *equivalent* w.r.t. f, written $c_P \approx_{\mathcal{H}} c_P$ (w.r.t. f), iff

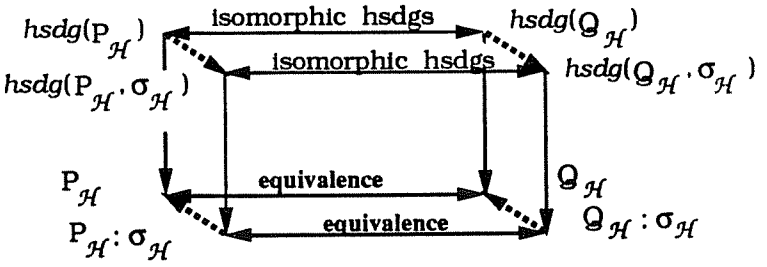
(1) neither c_P nor c_Q terminates successfully, or
(2a) c_P and c_Q both terminate successfully;
(2b) c_P and c_Q compute equivalent sequences of values at corresponding points (w.r.t. f); and
(2c) the final stores computed by c_P and c_Q are equivalent. □

THEOREM (***Pointer-Language Equivalence Theorem***). Let $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ be language $\mathcal{H}$ programs. Let *InSet* be a set of stores. Assume that P and Q have isomorphic *hsdgs* w.r.t. *InSet*. Let $\sigma_{\mathcal{H}}$ be a store in *InSet*. Then $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ are equivalent computations.

PROOF. If neither $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ nor $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$ terminates successfully, the theorem is immediate. Otherwise, assume that one of these programs—say, $P_{\mathcal{H}}$—terminates on $\sigma_{\mathcal{H}}$.

Let G_P and G_Q be the isomorphic *hsdgs* for $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ w.r.t. *InSet*. Since $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ exhibit strictly fewer traces w.r.t. $\sigma_{\mathcal{H}}$ than they do w.r.t. *InSet*, G_P and G_Q must also be *hsdgs* for $P_{\mathcal{H}}$ and $Q_{\mathcal{H}}$ w.r.t. $\sigma_{\mathcal{H}}$.

The Pointer-Language Equivalence Theorem is therefore equivalent to the assertion that (•2) the isomorphism of G_P and G_Q and the termination of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ imply the equivalence of $P_{\mathcal{H}}: \sigma_{\mathcal{H}}$ and $Q_{\mathcal{H}}: \sigma_{\mathcal{H}}$. The Pointer-Language Theorem therefore follows from the proof sketch for assertion (•2) given in Section 7.3.5, *Step 2*.



This concludes the proof of the Pointer-Language Equivalence Theorem. □

7.4. Practical Implications of the Pointer-Language Equivalence Theorem

The definition of language $\mathcal{H}$ given in Chapter 2 makes simplifying assumptions about freelists, procedure activation records, and atoms. This section considers how these assumptions affect the applicability of the result proved in the previous section.

7.4.1. Freelists

The assumption that pointer-language programs have unboundedly long freelists is comparable to the assumption, often made in optimizing compilers, that arithmetic can be reordered without causing arithmetic overflow. Ignoring the possibility of overflow allows useful optimizations to be performed that

---

Program MAX81:		Program MAX161:	
			parbegin
[1]	$a := P(1)$	[1]	$a := P(1)$
[2]	*display* (a)	[4]	$b := P(2)$
[3]	$a :=$ **nil**		**parend**
		[2]	*display* (a)
[4]	$b := P(2)$	[5]	*display* (b)
[5]	*display* (b)	[3]	$a :=$ **nil**
[6]	$b :=$ **nil**	[6]	$b :=$ **nil**

Function P ( ) allocates 80 structures, but has no other side effects.
Function *display* ( ) displays that part of the store that is referenced by its argument, but has no other side effects.

Programs MAX81 and MAX161 exhibit isomorphic sets of dependences. Programs MAX81 and MAX161, however, have different peak memory requirements. Program MAX81's store contains a maximum of 81 accessible structures. Program MAX161's store contains a maximum of 160 accessible structures. MAX81 will therefore succeed, and MAX161 fail, if an implementation has (*e.g.*) only 120 locations in its freelist.

Figure 7.9. Two programs with isomorphic *hsdgs* that return different results in the presence of a short freelist.

might not otherwise be possible. Consider, for example, the two programs depicted in Figure 7.9. If P is a time-consuming function call, then program MAX161 will run in considerably less time than program MAX81. Program MAX161, however, also has a higher peak demand for memory than program MAX81. Program MAX81 will therefore run to completion in some environments where program MAX161 fails.

Possible overflows that could arise from a reordering of a computation's statements are not accounted for the proof of the Pointer-Language Equivalence Theorem. The freelist problem has been sidestepped by assuming that a program's successful completion is never dependent on the reuse of previously allocated storage. Although anomalies of the kind described in the previous paragraph are unfortunate, it is hard to see how they can be prevented without inhibiting many important—and normally valid—optimizations. One possible solution to this problem is to let programmers specify that certain sections of a program should not be optimized. A different approach to guaranteeing the safety of program optimizations in the presence of heap allocation is discussed in [Cha88].

7.4.2. Procedure Activation Records

Treating a procedure activation record (PAR) as a heap-allocated structure simplifies the reduction by allowing a uniform treatment of allocatable structures. A more realistic, stack-based model of procedure evaluation would give a slightly less optimistic picture of a program's interprocedural dependences, at the cost of a considerably messier reduction.

The principle difference between the heap-based and stack-based models of procedure activation is that the stack contains reusable locations. The decision to place a PAR P in the heap, in effect, is a commitment that P's use of space will not conflict with any of the PARs that are created before or after it. Consider, for example, the following two-statement program:

[1] **call** $A(\cdots)$; [2] **call** $B(\cdots)$;

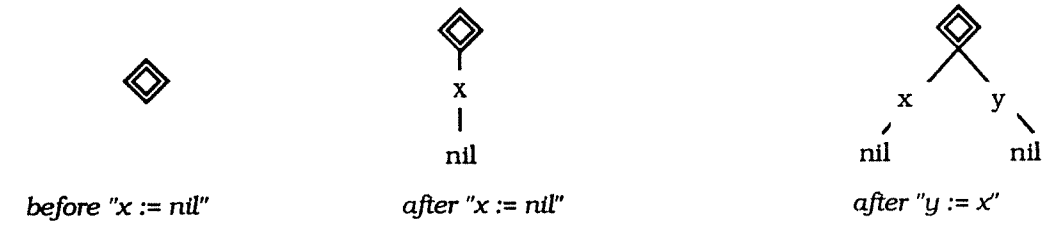
Assume that A and B are independent procedures. If A's and B's PARs are allocated in the heap, then A and B can be run in parallel. If A's and B's PARs are allocated in the stack, then statements that access A's PAR may be anti-dependent on the statement that invokes B.

7.4.3. Atoms

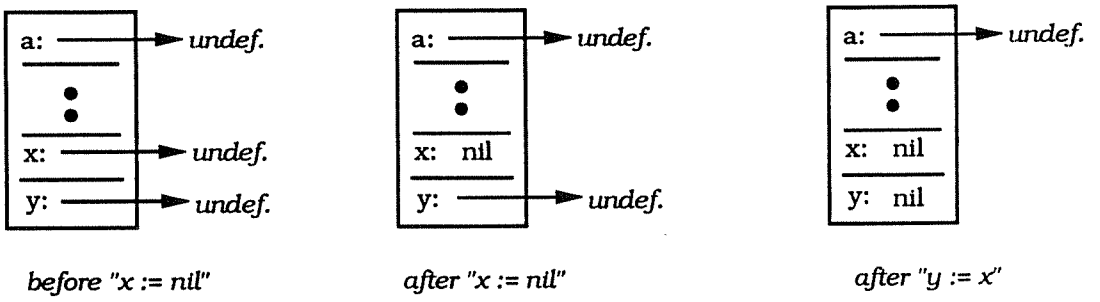
Figure 7.10 depicts three implementation techniques for atoms. The first technique, which is depicted in Figure 7.10(a), treats atoms as unshared, tagged structures that lack reference fields. The technique depicted in Figure 7.10(a) is the one modeled in this chapter.

A second representation of atoms is depicted in Figure 7.10(b). Here, the equations for structures have been rewritten so that fields have type $Loc + \text{ATOM}$, rather than type Loc. The representations depicted in Figures 7.10(a) and 7.10(b) are equivalent under the assumption that programs have unbounded freelists. Representation 7.10(a) was adopted in this thesis because it simplifies the presentation: it makes figures more compact, and semantic equations more uniform. Representation 7.10(b) would be preferred in an actual implementation, since it uses less space.

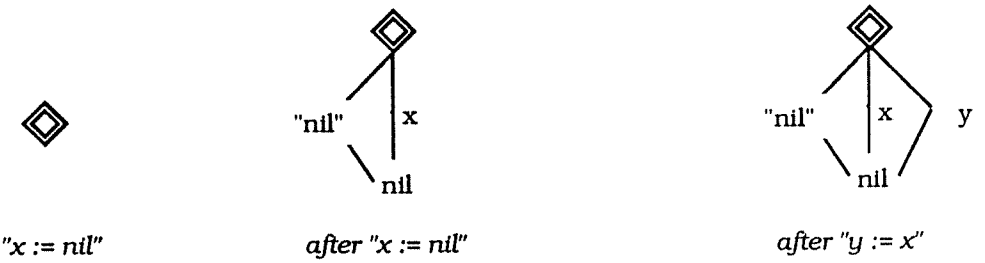
An alternative implementation of atoms, which treats atoms as *shared* objects, is depicted in Figure 7.10(c). Chase observes that realistic implementations of Lisp-like languages mix shared and unshared atoms [Cha88]. Indicator bits are typically set aside that specify whether a location contains a reference or



(a). Atoms as unshared structures



(b). Atoms as unshared values



(c). Atoms as shared structures

Figure 7.10. Three possible implementations of atoms.

a constant. Commonly used atoms like **nil** and small integers are typically stored as in Figure 7.10(b). Atoms that are too large to be stored in individual locations, such as strings and large numbers, are shared to conserve space.

Representation 7.10(a) was chosen over representation 7.10(c) to simplify the presentation. The decision to adopt the one representation over the other is significant, since the use of shared atoms complicates the proof of correctness. To see why, consider the following program, program *P*:

$[p]\ \ x := 10000\,;\ \ [q]\ \ y := 10000$

If atoms are represented as depicted in Figure 7.10(a), then statements p and q are independent. If atoms are represented as depicted in Figure 7.10(c), then q is flow-dependent on p. Statement p allocates a struc-

ture s that represents the value 10000, and statement q creates a reference to s.

Dependence $p \longrightarrow_f q$, however, is a needless constraint on the example program's evaluation. P is obviously equivalent to the program P' obtained by reversing p and q:

$$[q] \;\; y := 10000 \,; \;\; [p] \;\; x := 10000$$

To prove the equivalence theorem for an implementation of $\mathcal{H}$ with shared atoms, one could argue that P is equivalent to the following program:

$$[p'] \;\; struct_{10000} := 10000 \,; \;\; [p] \;\; x := struct_{10000} \,; \;\; [q] \;\; y := struct_{10000}$$

(and similarly for P'). The effort required to formalize this argument, however, does not seem commensurate with the benefits that would accrue from it.

7.5. Related Work

7.5.1. A brief history of *dbrs*

This section surveys earlier *dbrs*, and describes those *dbrs* that influenced the design of the *hsdg* in more detail.

7.5.1.1. The early history of *dbrs*

Dbrs have been in existence since 1972. The first *dbr*, the Kuck-Muraoka-Chen ***data dependence graph*** (***ddg***) [Kuc72], is a direct descendant of the Ramamoorthy-Gonzalez ***program graph*** [Gon69]. The program graph, which was developed in the late 1960's, is a directed acyclic graph, whose nodes represent either statements, or sets of statements, and whose edges represent evaluation constraints (*cf.* Figure 7.11). These evaluation constraints, roughly speaking, correspond to control, flow, anti-, and output dependences. The *ddg* differs from the program graph in the following two ways:

* *Ddgs* provide no information about a program's control structure. Kuck *et. al.* simplified the definition of their graph by stipulating that *ddgs* be used to represent structured programs.
* The program graph does *not* depict precedence constraints between individual statements in loops: the set of all statements in a given loop l are represented as a single node in a program graph. The *ddg*, on the other hand, supported loops *and* nested loops. Kuck *et. al.* also developed strategies for using *ddgs* to parallelize loops that access multi-dimensional arrays.

The version of the *ddg* described in [Kuc72] represents what Kuck, Muraoka, and Chen then referred to as a program's *forward data dependences*—and what would now be referred to as a program's flow and output dependences. Anti-dependences (there called *reverse dependences*) were eliminated by variable renaming. Subsequent work by the University of Illinois group formalized the notions of flow, output, input, and anti-dependence; these distinctions are made, for example, in Kuck's 1978 text on compiler construction [Kuc78]

Subsequent reports by the University of Illinois group used *ddgs* as a basis for program transformation. The efforts of the Illinois group focused primarily on parallelizing array-manipulating statements in nested loops. Figure 7.12, which gives an example *ddg*, is adapted from Wolfe's thesis [Wol82]. This *ddg* is annotated with information that characterizes loop-dependence interactions.

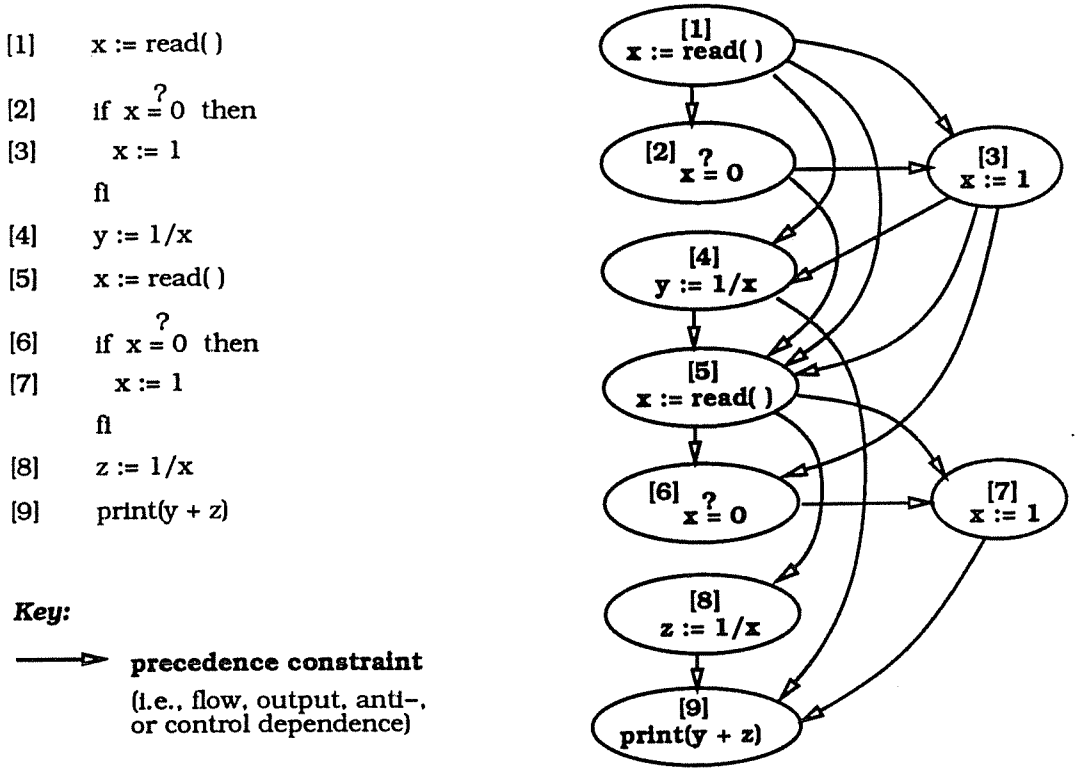


Figure 7.11. A Ramamoorthy-Gonzalez program graph.

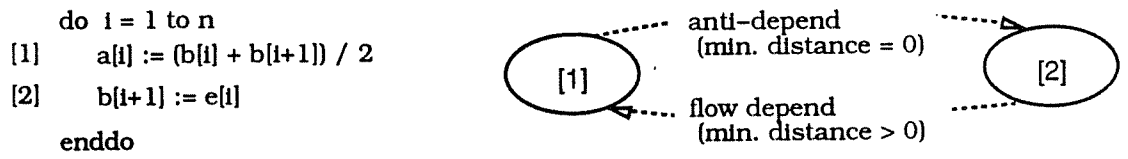


Figure 7.12. An example data dependence graph, together with information about a program's dependences.

The University of Illinois group used *ddgs* to optimize structured program fragments. A later paper by Allen, Kennedy, Porterfield, and Warren describes a simple program transformation that extends the set of programs that *ddgs* can support [All83a]. This transformation converts a fragment that contains "IF ··· GOTO" statements into an equivalent fragment whose "IF" statements guard structured blocks of code. *Ddg*-based optimizations can then be applied to the transformed fragment. A different approach to incorporating control-flow information into *dbrs* was developed by Ottenstein ([Ott78], cited in [Fer83]). Ottenstein's *dbr*, the ***data flow graph***, represented a program as a *pair* of graphs: a data dependency graph, together with its control flow graph.

References to other *dbrs* from the 1970s, including Dennis' work on dataflow machines and the Parafrase compiler, are given in the Ferrante, Ottenstein, and Warren report on *program dependence graphs* [Fer87].

7.5.1.2. Program dependence graphs

In 1982, Ferrante and Ottenstein (re)discovered that control and dependence information could be combined in a single *dbr*. The resulting *dbr*, the ***extended data flow graph*** (*edfg*), gives a self-contained picture of the dependences that constrain a program's evaluation [Fer83]. The *edfg*'s principle limitation is that programs must be structured. This limitation was subsequently removed by Ferrante, Ottenstein, and Warren, who named the resulting structure the ***program dependence graph*** (*pdg*) [Fer83a]. Figure 7.13 depicts a Ferrante-Ottenstein-Warren *pdg*; the depicted *pdg* shows a program's flow, output, and anti-dependences. Various advantages of *pdgs* (resp. *ddgs*) were cited by Ferrante, Ottenstein, and Warren, and in a companion paper by Ottenstein and Ottenstein [Ott84]; most of these advantages are corollaries of the observation that *pdgs* give a self-contained characterization of a program's behavior.

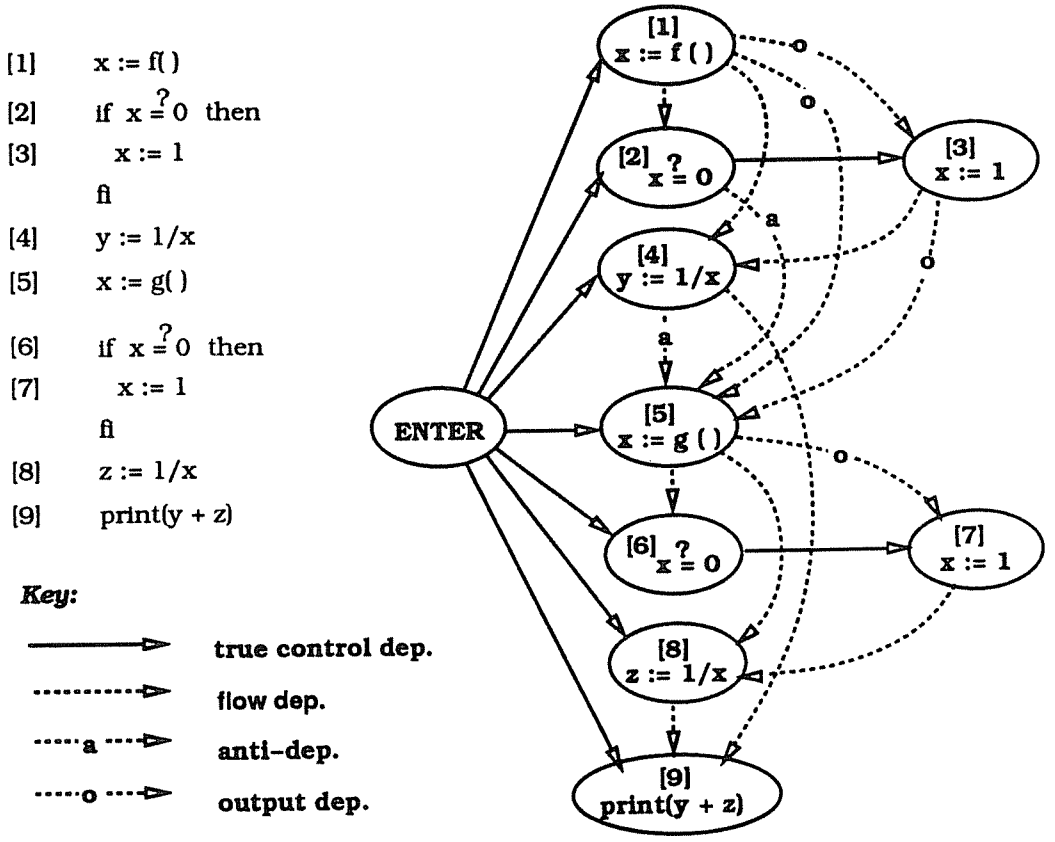


Figure 7.13. A Ferrante-Ottenstein-Warren *pdg*.

A subsequent paper by Ferrante, Ottenstein, and Warren describes two additional kinds of *pdgs* [Fer87]. The first uses explicit *load and store operators* to model how programs alter internal state. This idea was later rediscovered by Pingali *et. al.*, who use loads and stores in a dataflow-graph-like *dbr* (*cf.* § 7.2.4). The second, the ***hierarchical pdg***, gives a structured picture of a program's dependences. A hierarchical pdg for a program P is a collection of *pdgs* $P_1 \cdots P_n$ that represent different fragments of P. Each P_i contains a head vertex that represents that fragment's overall behavior. Edges that are local to the P_i represent intra-fragment dependences. Edges between the heads of the different P_i's represent inter-region dependences. Ferrante *et. al* observe that hierarchical pdgs can be used for vectorization and loop fusion.

Horwitz, Prins, and Reps adjusted the edge set of Ferrante *et. al.*'s *pdg*, replacing anti- and output dependences with what was then a new kind of dependence—the *def-order* dependence (*cf.* Figure 7.1) [Hor87, Hor89]. The resulting *pdg* gives a better characterization of a program's *slices*—*i.e.*, its logically related sets of statements (*cf.* Chapter 3, §7.2.5). Horwitz *et. al.* also added two new types of vertices to a *pdg's* vertex set. The one, the *initial definition* vertex, was added to account for variables that were referenced before being used. The other, the *final use* vertex, allows programs to be analyzed w.r.t. the final values of specially selected variables (*cf.* Figure 7.14).

7.5.1.3. Def-order-dependence-free *dbrs*

Sections 7.5.1.3, 7.5.1.4, and 7.5.1.5 discuss successors of the *pdg*. One reason for the continuing interest in new *dbrs* is the search for simpler, more elegant characterizations of program behavior. The use of def-order dependences to characterize program behavior, for example, has the following drawbacks:

- Def-order dependences, like output dependences, arise from the reuse of locations. They do not reflect a true sharing of information between statements.
- Def-order dependences are created by the interaction of *three* program points. This makes it harder to tell whether one program point is def-order-dependent on a second. More specifically, if P is a program, and A is a fragment of P that contains two program points p and q, then an analysis of A alone might not reveal whether q is def-order-dependent on p w.r.t. P.
- Def-order dependences can proliferate in programs with multiply-defined variables. Figure 7.15 depicts a program that has $O(n)$ assignment statements and $O(n^2)$ def-order dependences.

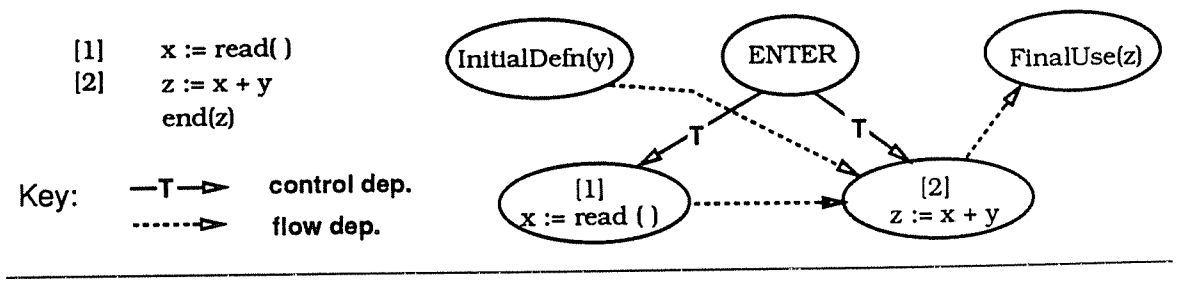


Figure 7.14. Initial definition and final use vertices.

Three techniques have been proposed for creating def-order-dependence-free *dbrs*. The first technique, developed by Alpern, Wegman, and Zadeck, uses assignment statements and variable renaming to eliminate a program's output and anti-dependences [Alp88]. Alpern *et. al.*'s algorithm for eliminating output and anti-dependences assumes that different definitions of a variable x that reach the same program point lie along different paths in a program's control-flow graph. The technique first places the statement "$x := \phi(x, x)$" at each of x's ***join birthpoints***—*i.e.*, at those program points where different definitions of x first converge [Rei81]. The placement of ϕ nodes ensures that only one definition of a variable x is live along any segment of a control-flow graph. Every occurrence of x is then subscripted with an index that pairs the occurrence with the assignment statement that defines its value. The application of Alpern *et. al.*'s construction to an example program is illustrated in Figure 7.16

The remaining two techniques are essentially variants of the Alpern-Wegman-Zadeck ϕ node. Felleisen and Cartwright [Car89] use ***valve nodes*** to eliminate a program's def-order dependences. Valve nodes are assignment statements that block definition-free paths in control-flow graphs. Specifically, a valve node "$x := x$" is added to a path π in a program's control-flow graph when (1) π does not contain an assignment to x, and (2) a parallel path does. Figure 7.17 illustrates the placement of a valve node in an example program.

The remaining technique, which was developed by Yang, Horwitz, and Reps, uses a different kind of ϕ node to render def-order dependences redundant [Yan89, Yan90]. The Yang-Horwitz-Reps ϕ node is an assignment statement "$\phi_{label}: x := x$" that is placed at a variable x's join birthpoint. The *label* subscript identifies the syntactic construct that created the birthpoint. Figure 7.18 illustrates the placement of a ϕ_{if}

---

[1] $x := 1$
if $pred_2$ **then** [2] $x := 2$ **fi**
if $pred_3$ **then** [3] $x := 3$ **fi**
...
if $pred_n$ **then** [n] $x := n$ **fi**
[n+1] $y := x$

This program exhibits one def-order dependence
$j \longrightarrow_{do([n+1])} k$
for every j and k such that $1 \leq j < k \leq n$.

Figure 7.15. A program that has $O(n)$ assignment statements and $O(n^2)$ def-order dependences.

	Before ϕ–node placement	After ϕ–node placement
[p]	$x := 0$	$x_1 := 0$
	if *pred* **then** [q] $x := 1$ **fi**	**if** *pred* **then** $x_3 := 1$ **fi**
		$x_{3,5} := \phi(x_1, x_3)$;
	$y := x$	$y_4 := x_{3,5}$

Figure 7.16. Alpern-Wegman-Zadeck ϕ-node placement in an example program.

Before valve–node placement		After valve–node placement	
[1]	$x := 0$	[1]	$x := 0$
[2]	**if** *pred* **then** [3] $x := 1$ **fi**	[2]	**if** *pred* **then** [3] $x := 1$ **else** [3.5] $x := x$ **fi**
[4]	$y := x$	[4]	$y := x$

Figure 7.17. Valve-node placement in an example program. Statement [3.5] is the valve node.

node in an example program.

The Yang-Horwitz-Reps ϕ node renders a dependence $p \longrightarrow_{do(r)} q$ redundant by fixing the relative execution order of p and q. Each ϕ node has two incoming flow dependences. These dependences are paired by the ϕ node with different paths in a program's control flow graph. This pairing of dependences with graphs, when combined with information about a program's syntactic structure, fixes the relative execution order of p and q.

Yang *et. al.* refer to their def-order-edge-free *dbr* as a ***program representation graph*** (*prg*). Figure 7.19 shows that the number of components in a program's *prg* may be asymptotically of lower order than the number of components in its *pdg* (*i.e.*, $O(n)$ vs. $O(n^2)$).

7.5.1.4. Interpretable *dbrs*

A second reason for the continued interest in new *dbrs* is the search for *dbrs* that can be executed efficiently. This quest is motivated, in part, by the observation that *dbrs* expose a program's potential parallelism. This quest is also motivated by the observation that *pdgs* do not appear to be a good starting point for dataflow-style program execution. Selke [Sel90] has given a graph-rewriting rewriting semantics for *pdgs*. Although this semantics is useful for reasoning about how dependences constrain program execution, it has the following important limitation: useless, intermediate values of a variable x may be propagated to a statement r before r gets the proper value of x. Consider, for example, the following program fragment:

Before ϕ–node placement		After ϕ–node placement	
[1]	$x := 0$	[1]	$x := 0$
[2]	**if** *pred* **then** [3] $x := 1$ **fi**	[2]	**if** *pred* **then** [3] $x := 1$ **fi**
		[3.5]	$\phi_{if} : x := x$
[4]	$y := x$	[4]	$y := x$

Figure 7.18. ϕ-node placement in an example program.

Effect of inserting ϕ nodes into program in Figure 7.16

Before ϕ–node insertion	After ϕ–node insertion
[1] $x := 1$	[1] $x := 1$
if $pred_2$ **then** [2] $x := 2$ **fi**	**if** $pred_2$ **then** [2] $x := 2$ **fi**; [2.5] ϕ_{if}: $x := x$;
if $pred_3$ **then** [3] $x := 3$ **fi**	**if** $pred_3$ **then** [3] $x := 3$ **fi**; [3.5] ϕ_{if}: $x := x$;
...	...
if $pred_n$ **then** [n] $x := n$ **fi**	**if** $pred_n$ **then** [n] $x := n$ **fi**; [n.5] ϕ_{if}: $x := x$;
[n+1] $y := x$	[n+1] $y := x$

The program on the left can be represented by a *pdg* that has $O(n^2)$ elements: $O(n)$ vertices and flow-dependence edges, and $O(n^2)$ def-order dependences.

The updated program can be represented by a *prg* that has $O(n)$ vertices and flow dependences.

Figure 7.19. Using ϕ nodes to reduce the size of a *dbr*.

[p] $x := 10$; **if** *pred* **then** [q] $x := 20$ **fi**; [r] **print**(x);

If *pred* is true, then two values of x propagate to r: 10, which should not be printed, and 20, which should.

Ramalingam and Reps were among the first to argue that the program representation graph (*prg*) (*cf.* §7.2.3) constitutes a good basis for dataflow-style program execution [Ram89]. This report, which gives a dataflow-like semantics for *prgs*, also discusses the limitations of using *pdgs* to execute programs.

A second dataflow-like *dbr* was developed by Ballance, Maccabe, and Ottenstein [Bal90]. This *dbr*, which Ballance *et. al.* call the ***program dependence web*** (***pdw***), uses Alpern-style ϕ nodes to eliminate output dependences. An interesting feature of the program dependence web is that it can be interpreted in a control-driven, data-driven, or demand-driven fashion.

Pingali, Beck, Johnson, Moudgill, and Stodghill describe a third type of interpretable *dbr*, the ***dependence flow graph*** [Pin91]. Dependence flow graphs are *pdw*-like *dbrs* that support explicit load and store instructions. Dependence flow graphs also incorporate a notion of dependence that the authors refer to as *imperative* dependence. Intuitively, an imperative dependence is exhibited by a pair of statements like "*load* x ; *test* x"; the test cannot proceed until the load of x is complete.

7.5.1.5. System dependence graphs

A third reason for the continued interest in new *dbrs* is the search for *dbrs* that support more complex languages. *Pdgs*, for example, only model *intra*procedural aspects of program evaluation. The first *dbr* for languages with procedures, the ***system dependence graph*** (***sdg***), was developed by Horwitz, Reps, and Binkley in 1988 [Hor88a, Hor90a]. The *sdg* is an enhanced *pdg* that uses new kinds of edges and vertices to represent call statements and procedures. The model language that Horwitz *et. al.* use is a simple, structured language. It supports scalar-valued variables, **if** and **while** statements, and the following, four-step, *value-result* protocol for parameter passing:

1. When a procedure B is called from procedure A, all non-local variables referenced or modified by B are first copied into a special input buffer, ι.
2. When control is first transferred from A to B, the initial values of B's formal parameters and non-local variables are obtained from ι.
3. When B finishes evaluating, the final values of all non-locals modified during the evaluation of B are written into a special output buffer, ω.
4. Control is then returned to A, which uses ω to update its copy of every variable altered by B.

The example language that Horwitz *et. al.* assume simplifies the task of characterizing a program's interprocedural evaluation. In languages that lack dynamic allocation, it is possible to identify a finite set of variables that may be read or written during a call to a procedure A—that is, by the body of A proper, or by procedures that might be invoked during A's evaluation. Horwitz *et. al.* use this observation to control how a program's interprocedural dependences are represented. Let A and B, for example, be two procedures in a program P. If A does not contain a statement that calls B, then P's *sdg* lacks edges of the form (*ap*, *bp*), where *ap* and *bp* are points in A and B, respectively. If A, on the other hand, calls B, then *every* interprocedural data dependence that arises from a call on B is of the form (*ap*, *bp*), where *ap* is a special program point that implements a call to B, and *bp* is a special program point that (intuitively) initializes or finalizes B's procedure activation record. Furthermore, *every* interprocedural control dependence that arises from a call on B runs from the point that represents the call to B's entry point. This technique for representing interprocedural dependences, which will be referred to here as ***dependence encapsulation*** (or simply encapsulation), yields a *dbr* that gives a good characterization of how procedures interact. Encapsulation also allowed Horwitz *et. al.* to develop an efficient interprocedural slicing algorithm—one that uses special edges to bypass parts of the *sdg* during slice computation (see below).

An *sdg* for a program P, S_P, is a collection of smaller *dbrs* that represent P's procedures, linked by edges that represent P's interprocedural dependences. Graph S_P contains one *dbr* for each of P's procedures. One of these *dbrs*, the ***distinguished procedure dependence graph*** (distinguished πdg) depicts P's main procedure. The remaining *dbrs*, called ***procedure dependence graphs*** (πdgs), depict P's auxiliary procedures—procedures that are called by *main*. (*N.B.*: the name πdg is used to distinguish the procedure dependence graph from the similarly-named program dependence graph.)

A *procedure dependence graph* (πdg), roughly speaking, is an extended *pdg* that models procedure entry and exit and supports call statements. πdgs and *pdgs* both contain

- an entry vertex that corresponds to the initial locus of control;
- vertices that correspond to predicates and assignment statements;
- initial definition vertices;
- final use vertices; and
- edges that represent control, flow, and def-order dependences.

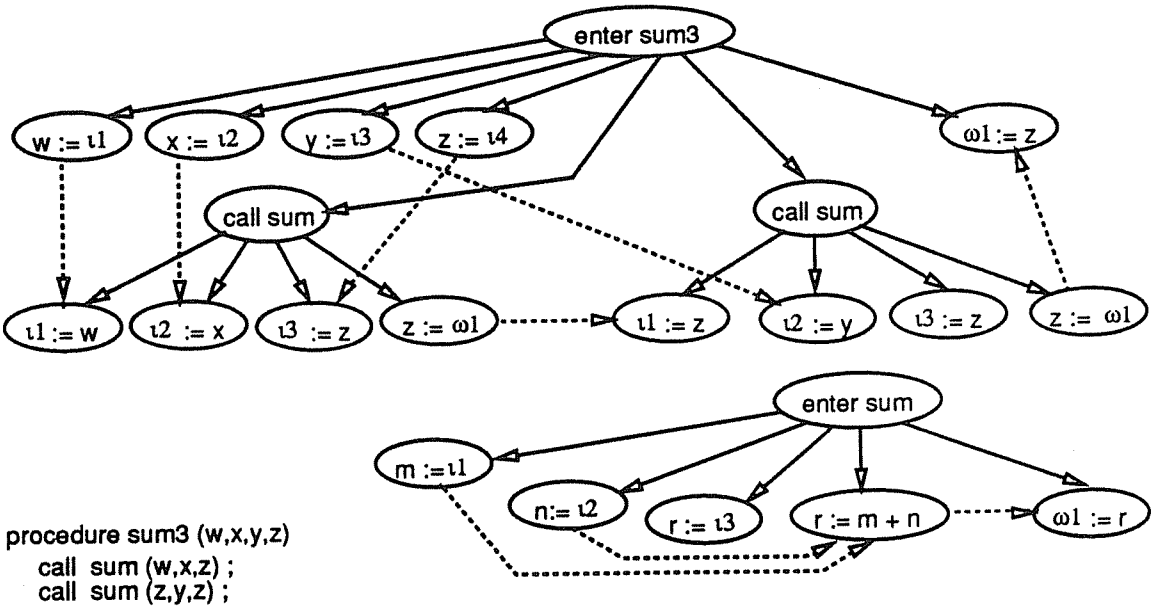
A πdg also contains five new kinds of vertices. Three of these vertices represent call statements. A ***call vertex*** depicts a procedure call *per se*. An ***actual-in*** vertex, which depicts the initial write of a value into buffer ι, models Step 1 in the value-result protocol. An ***actual-out*** vertex, which depicts a final acquisition of a value from ω, models Step 4 in the protocol. The other two kinds of vertices model parameter passing from the callee's point of view.

- ***Formal-in vertices***, which are analogous to initial definition vertices, model Step 2 in the calling protocol. More specifically, let P be a procedure, and Π_P its πdg. Then Π_P has one formal-out vertex for every formal parameter and non-local variable that could be modified during a call to P.
- ***Formal-out vertices***, which are analogous to final use vertices, model Step 3 in the calling protocol. More specifically, let P be a procedure, and Π_P its πdg. Then Π_P has one formal-in vertex for every formal parameter and non-local variable that could be read during a call to P; that is, every variable that could be read by a statement in P proper, or by a statement in a procedure that P calls.

Two example πdgs are depicted in Figure 7.20; variables of the form "ιn" and "ωn" represent slots in the transfer buffers.

A *distinguished* πdg is an extended *pdg* that models a program's main procedure. Figure 7.21 depicts an example *distinguished* πdg.

πdgs are linked by two kinds of edges that represent a program's interprocedural dependences. The first, the interprocedural control dependence edge, links a call statement with the entry vertex of the called procedure. The second, the interprocedural flow dependence edge, links actual-in vertices to formal-in ver-



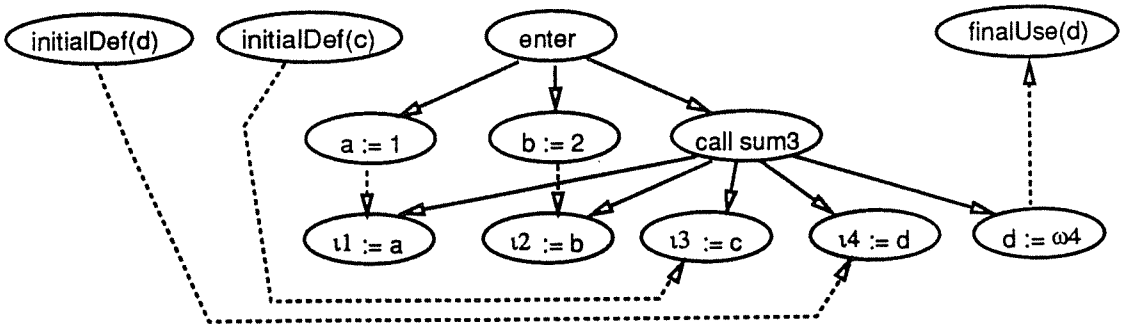

```
procedure sum3 (w,x,y,z)
   call sum (w,x,z) ;
   call sum (z,y,z) ;
return

procedure sum (m,n,r)
    r := m + n ;
return
```

Procedure dependence graphs for sum3, sum.

- Solid arrows denote control dependences
- Dotted arrows denote flow dependences
- Iotas and omegas are interprocedural linkage variables

Figure 7.20. Example procedure dependence graphs.




```
procedure main ( )
   a := 1
   b := 2
   call sum3 ( a, b, c, d)
end (d)
```

Distinguished procedure dependence graph for procedure main.

- Solid arrows represent true-valued control dependences
- Dotted arrows represent flow dependences
- Iotas and omegas are interprocedural linkage variables

Figure 7.21. A distinguished procedure dependence graph.

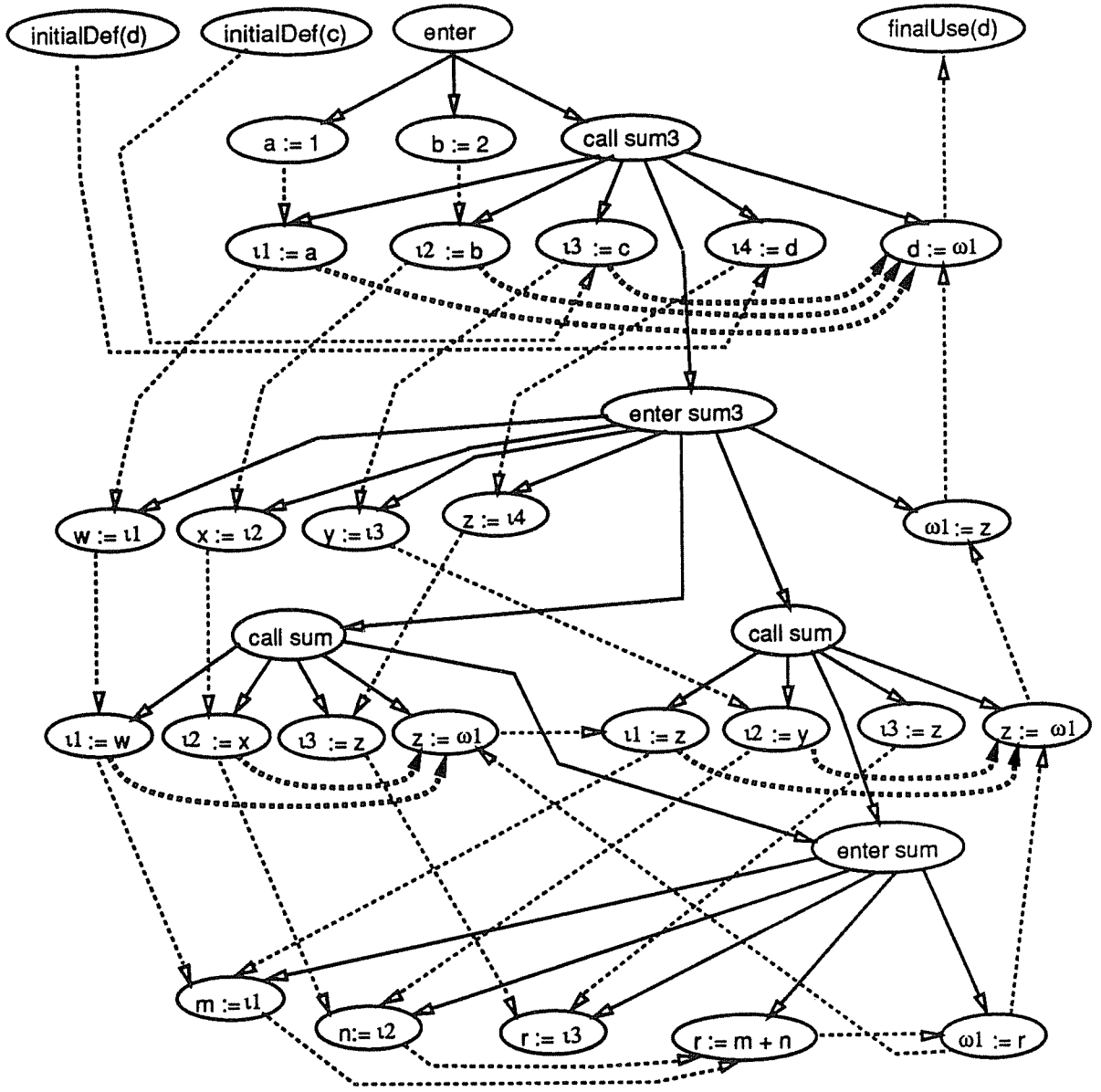
tices, and formal-out vertices to actual-out vertices. A third kind of edge, the ***summary edge***, is a convenience edge that simplifies the computation of a program's slices. Let a_{in} and a_{out}, for example, be actual-in and actual-out vertices associated with a call to a procedure B. If B uses the value of a_{in} to compute the value of a_{out}, then the *dbr* that contains a_{in} and a_{out} has an interprocedural summary edge e from a_{in} to a_{out}. This edge represents a transitive flow dependence from a site that defines an input parameter to a site that uses it. Edge e allows the contribution to a slice made by a call on a procedure B to be determined *without* examining B. Horwitz *et. al.* use summary edges to obtain a polynomial-time algorithm for interprocedural slicing that is (1) more precise than the original algorithm given by Weiser [Wei84], and (2) as precise as, but more efficient than, a subsequent algorithm given by Hwang, Du, and Chou [Hwa88].

Figure 7.22 gives a complete picture of an example *sdg*.

7.5.2. Previous soundness theorems for *dbrs*

Horwitz, Prins, and Reps were the first to investigate whether dependence graphs provide an adequate representation of a program's semantics [Hor88]. Horwitz *et. al.* proved that programs with isomorphic *pdgs* computed identical final stores, relative to a structured language with scalar variables. Reps and Yang strengthened this result by showing that terminating programs with isomorphic *pdgs* computed identical sequences of values at corresponding program points [Rep89]. A second proof of the Equivalence Theorem that develops a graph-rewriting semantics for *pdgs* was given by Selke [Sel89]. This work was later extended by Selke to obtain a comparable theorem for dynamic *pdgs* [Pfe91a].

Reps and Yang were the first to investigate the semantics of program slicing. In [Rep89], Reps and Yang showed that *pdgs* provide an adequate characterization of a program's slices, relative to a structured language with scalar variables. A second proof of the Slicing Theorem has been given by Selke [Sel90]. This work was later extended by Selke to obtain a comparable theorem for dynamic *pdgs* [Pfe91a].




```
procedure main ( )
   a := 1
   b := 2
   call sum3 ( a, b, c, d)
end (d)
```

```
procedure sum3 (w,x,y,z)
   call  sum (w,x,z)
   call  sum (z,y,z)
return
```

```
procedure sum (m,n,r)
   r := m + n
return
```

Figure 7.22. Example system dependence graph. Dashed lines with solid arrowheads are interprocedural flow dependence summary edges.

The first proof of an equivalence theorem for a language with heap allocation and pointer variables was given by Pfeiffer and Selke [Pfe91a]. That paper proves an equivalence theorem for a procedureless subset of $\mathcal{H}$. The *dbr* that was used to represent programs in this report, the *hpdg*, is essentially an *hsdg* that lacks support for procedures and call statements. This report also showed that an *hpdg* gave an adequate characterization of a program's *slices* (*cf.* §7.1). The proof strategy used in [Pfe91a] is similar, but not identical, to the one adopted in this thesis. The reduction defined in the earlier report eliminates most—but not all—of a program P's freelist-mediated dependences by maintaining a separate freelist at each of P's program points. More precisely, it eliminates all freelist-mediated dependences of the form $p \longrightarrow_f q$, where p and q are different program points. This partitioning, however, does not eliminate freelist-mediated dependences of the form $p \longrightarrow_f p$, where p (*e.g.*) is a point inside a loop that allocates a structure. Intuitively, such dependences persist because every point p is associated with its own fragment of the freelist; the location allocated by the kth evaluation of a point p is therefore dependent on the location allocated by the $k-1$st. The reduction given here removes this restriction by first unrolling a program—thereby ensuring that every allocation site in the reduced program evaluates no more than once.

Other work on the semantics of dependence-graph representations include

- Yang's thesis, which demonstrates the soundness of a *dbr*-splicing operation known as *program integration* [Yan90];
- Selke's program-transformation calculus for *pdgs* [Sel90a];
- a semantics for *prgs*, developed by Ramalingam and Reps [Ram89];
- Pingali *et. al.*'s soundness of representational soundness for the *dependence-flow graph* [Pin91]; and
- a report by Binkley, Horwitz, and Reps, which proves an equivalence theorem for *sdgs* [Bin89]. Binkley *et. al.*'s proof of the *sdg* Equivalence Theorem, which reduces two programs with isomorphic *sdgs* to two programs with isomorphic scalar *pdgs*, inspired the approach to proving facts about *hsdgs* used here.

7.6. The Limitations of the *Hsdg*

The *hsdg* fails to incorporate three recent ideas that have been used to improve the successors of the *pdg*:

1. The *hsdg* uses a program's def-order dependences to depict its behavior.
2. The *hsdg* provides a non-encapsulated characterization of a program's dependences: not all interprocedural dependences are captured by nodes that represent the interface between caller and callee.
3. *Hsdgs* do not use distinct vertices to model distinct values in a program's initial and final stores.

This section explains how dynamic allocation and reference variables complicate the task of defining a *dbr* that meets these three goals. The final two sections also propose ideas for future research—speculative suggestions for developing a *dbr* that does not suffer (to the same extent) from limitations 1 and 2.

7.6.1. ϕ nodes vs. def-order dependences

The opening of Section 7.5.1.3 argued that it was advantageous to develop *dbrs* that do *not* use def-order and output dependences to model a program's behavior. Section 7.5.1.3 also sketched three alternatives to using def-order dependences to model program behavior. All three techniques, unfortunately, assume that if $p \longrightarrow_{do} q$ is a def-order dependence, then p and q must lie in distinct basic blocks: *i.e.*, that either p

```
        procedure main()                    procedure A(ref i, ref j)
[1]     call A(a, b);               [p]     i := 1;
[2]     call A(a, a);               [q]     j := 2;
        return                      [r]     print(i)
                                            return
```

This program exhibits the def-order dependence $p \rightarrow_{do(r)} q$. The first evaluation of r prints the value defined by statement p; the second evaluation of r prints the value defined by statement q.

Figure 7.23. A program where pass-by-reference parameters give rise to def-order dependences in straight-line code.

must fail to dominate q, or q must fail to post-dominate p (*cf.* §3.4.2). This assumption, however, is *not* true of a language in which the locations that two statements name can change, relative to one another. If a procedure P, for example, has reference parameters, then two calls to P that create different aliases can create def-order dependences in straight-line code. An example of a *straight-line def-order dependence* is illustrated in Figure 7.23. In the first evaluation of A, in which x and y are not aliased, statement r reads the value that p assigns to x. In the second evaluation of A, in which x and y are aliased, statement r reads the value that q assigns to x.

One technique for eliminating straight-line def-order dependences is suggested by Ballance, Maccabe, and Ottenstein [Bal90]. Ballance *et. al.* propose that procedure calls that exhibit different aliasing patterns be treated as calls to different procedures Assume, for example, that procedure A has n reference parameters. Ballance *et. al.* create a distinct copy of P for every possible aliasing pattern that a call to A could exhibit. Each of the different copies of A is then analyzed separately, under a different assumption about A's aliases. Figure 7.24 illustrates their proposed workaround for the program in Figure 7.23.

A major disadvantage of this technique is the potential for code explosion. Code replication might be practical when there are only a few variations in a program's aliasing patterns. Code replication, however,

```
        procedure main()            procedure A1(ref i, ref j)          procedure A2(ref i, ref j)
                                    assert i ≠ j;                       assert i ~ j;
[1]     call A1(a, b);      [p]     i := 1;                     [p]     i := 1;
[2]     call A2(a, a);      [q]     j := 2;                     [q]     j := 2;
        return              [r]     print(i)                    [r]     print(i)
                                    return                              return
```

The expression $i \sim j$ means that variable i is aliased to variable j; *i.e.*, that i and j denote the same location.

Figure 7.24. The Ballance-Maccabe-Ottenstein technique for eliminating def-order dependences in straight-line code. The program depicted above is a revised version of the one depicted in Figure 7.17.

appears impractical for pointer programs, since pointer programs can exhibit a myriad of aliasing patterns. This observation implies that any attempt to develop a def-order-dependence-free *dbr* for pointer languages will have to handle dependences that arise in straight-line code.

7.6.2. Why *hsdgs* aren't encapsulated *dbrs*

Dynamic allocation also complicates the development of an encapsulated *dbr* for language $\mathcal{H}$. To understand why this is so, consider how dynamic allocation complicates the task of describing the following procedure's behavior:

```
      procedure copy(p, q)
      local lp, lq;
[1]   lp := p;
[2]   lq := q;
[3]   while pred do  [4] lp.hd := lq.hd; [5] lp := lp.tl; [6] lq := lq.tl  od
      return
```

Since statements [5] and [6] are embedded in a loop, no *a priori* bound can be imposed on the number of structures that *copy* might access. This observation implies that a π*dg* that names every structure that *copy* might access could contain infinitely many formal-in and formal-out vertices.

It *should* be possible to define a π*dg* that gives an *approximate* characterization of the set of structures that *copy* manipulates. More specifically, it should be possible to extend the definitions of actual-in, actual-out, formal-in, and formal-out vertices to obtain new vertices that transfer *sets* of values across procedure boundaries. These vertices—call them ***approximate transfer vertices***—would use regular expressions to name the potentially infinite sets of values transferred between caller and callee. Procedure *copy*, for example, might be depicted as a π*dg* that has two formal-in vertices, four approximate formal-in vertices, and one approximate formal-out vertex:

1. One formal-in vertex that initializes p.
2. One formal-in vertex that initializes q.
3. One formal-in vertex that initializes $p.(tl)^*.hd$.
4. One formal-in vertex that initializes $p.(tl)^*.tl$.
5. One formal-in vertex that initializes $q.(tl)^*.hd$.
6. One formal-in vertex that initializes $q.(tl)^*.tl$.
7. One formal-out vertex that finalizes $p.(tl)^*.hd$.

The practical realization of this idea, however, requires more thought than it can be given at this time. A second feature of language $\mathcal{H}$, the alias-updating assignment statement, complicates the choice of a good set of approximate parameter-transfer vertices for a procedure like *copy*. A poor choice of approximate transfer vertices can yield an excessively pessimistic characterization of a procedure's semantics. This would be true, for example, of vertex 7 (above) when the loop at statement [3] never evaluates more than twice. A poor choice of approximate transfer vertices can also yield an excessively optimistic (*i.e.*, incorrect) characterization of a procedure's semantics. This would be true, for example, of vertex 7 when q is initially aliased to a location in the list headed by p.

7.6.3. Why *hsdgs* have one initial definition and no final use vertices

The presence of one initial definition in *hsdgs* is another concession to the presence of aliases in pointer languages. Since language $\mathcal{H}$ places no restrictions on the aliases in the initial store, it cannot be determined whether two identifier expressions initially denote the same object.

Final-use vertices were omitted from *hsdgs* to simplify the presentation. If an **end**($\cdots$) statement were added to language $\mathcal{H}$'s definition, then a program's n final-use vertices would be immediately subordinate to that program's **enter** vertex. The effect of final-use vertices could be also obtained in a slightly enhanced version of language $\mathcal{H}$ that supports print statements (*cf.* Chapter 8).

OTHER REMARKS ABOUT CHAPTER 7

It seems reasonable to ask whether it is possible to produce a more elegant proof of the Pointer-Language Equivalence Theorem. Binkley and Selke, for example, prove theorems about semantic properties of *dbrs* by first unfolding a program *irrespective* of a computation's initial store, and *then* reasoning about how the resulting infinite program evaluates w.r.t. a given initial store [Bin91, Sel90a]. The unfolding transformations used by Binkley and Selke, however, exhibit the following, important property: a *single* unfolding of a given syntactic construct (*e.g.*, a *while* loop) yields a new program that has a bounded number of program points. This is true in part because the languages considered by Binkley and Selke limit the number of distinct locations that any occurrence of a given program point can access. There is, however, no such *a priori* bound on language $\mathcal{H}$'s name space; a *single* unfolding of a statement that contains a selector access expression w.r.t. an infinite name space, for example, would yield a new program with an *infinite* number of program points.

The main reason for using language $\mathcal{S}$ to reason about pointer-program *dbrs* was that this allowed earlier results about *pdgs* to be applied to the study of pointer-language *dbrs*. This ability to appeal to earlier theorems about *pdgs*—in particular, Selke's version of the Equivalence Theorem—greatly simplified the task of proving theorems about *hsdgs*. There are, on the other hand, two aesthetic objections that can be raised against this use of language $\mathcal{S}$. The first is that the map from language $\mathcal{H}$ to language $\mathcal{S}$ makes the proof rather complicated. The second objection is that the very idea of a reduction is somewhat distasteful: it should be possible to reason about $\mathcal{H}$ directly, without recourse to an auxiliary language.

The author has not thought very much about alternative strategies for proving theorems about *hsdgs*. It may be possible, however, to dispense with the reduction by developing a graph-rewriting semantics for language $\mathcal{H}$. The postulated rewriting semantics would be an extension of the *pdg*-rewriting semantics developed by Selke [Sel89]. Selke's semantics evaluates a vertex like "$x := y$" by propagating an updated value of x to the vertices that are dependent on this vertex. The *hsdg*-rewriting semantics, on the other hand, would evaluate assignment statements by propagating an updated *fragment* of a store graph to a node's successor nodes. The tricky thing about developing such a semantics is that it would be harder to characterize exactly how one node affects another: to show, for example, that the resulting semantics is confluent.

8. A FEW CONCLUDING REMARKS

> *[Designing software is] like a Russian doll. Every time we finally crack open one problem, we find there's another one inside.* —M. Kapor [Wall Street Journal, May 11, 1990]

> *If a writer has chosen to be silent on one aspect of the world, we have the right to ask him: Why have you spoken of this rather than that? And since you speak in order to make a change, since there is no other way you can speak, why do you want to change this rather than that?*
> —J.-P. Sartre, cited in [Kau63]

During the nine months that led up to the completion of this thesis, the author discovered how much of a gap there can be between a collection of related results and a well-rounded theory. The principal contributions of this thesis, which are given below, are arguably a solid contribution to the literature on dependence analysis:

- Chapter 3 develops an alternative definition of def-order dependence: one that is more suitable for pointer-program analysis than the existing definition of def-order dependence.
- Chapters 3 though 6 develop a broad-based approach for analyzing a pointer program's data dependences—one that separates the mechanism for manipulating abstract states from the policy used to ensure that analyses terminate.
- Chapters 4 through 6 demonstrate that this approach is safe w.r.t. the example language's implementation semantics.
- Chapter 6 sets forth a new scheme for categorizing the various approaches to store approximation, together with proposed extensions for making *k*-limiting practical.
- Chapters 7 defines a new *dbr* for the example pointer language considered in this thesis, and demonstrates that this *dbr* gives a sound characterization of a program's meaning.
- Chapter 7 also demonstrates the importance of a new approach for arguing about *dbrs*—one that uses observations about a program's *actual* executions (rather than control-flow-graph-based estimates of its executions) to understand its behavior.

Chapters 2 through 7, however, also point out many important limitations of the theory developed in this thesis. The first of these limitations, which is mentioned in Chapter 2, is the omission of various ***common operators and constructs*** from the example language.

Input and output were omitted from the example language to simplify the discussion of data dependence. Recall that Chapter 3 uses *run-time behavior* as a basis for determining a program's heap-mediated data dependence. This notion of dependence is used because the semantics of pointer assignment makes naive, control-flow-graph-based estimates of heap-mediated dependence unattractive (§3.4.8). Control-flow-graph-based estimates of a program's dependences, on the other hand, seem a more natural starting point for estimating how programs manipulate input and output. The introduction of a second style of definition into the thesis, unfortunately, would have complicated the presentation.

Arrays are not considered in this thesis because the computation of array-mediated data dependence is a challenging subject in its own right; this area has already been, and continues to be, an object of extensive study (see, *e.g.*, [Wol91]). Also, Selke has shown that *pdgs* provide a sound model of programs that use arrays; this proof will be presented in her forthcoming thesis [*private communication*].

Goto statements are not considered in this thesis for two reasons. The first is that Selke also intends to argue that *pdgs* can be used to model programs in languages that contain these constructs. A second reason for not considering gotos is that use of *nonreducible* (*i.e.*, multiple-entry point) loops (*cf.* [Aho86]) complicates the notion of a carrier.

The analysis of ***higher-order procedures*** is another challenging topic of research. Reports by Harrison and Deutsch cited in Chapter 5 propose techniques for analyzing programs in languages that support higher-order procedures. Another interesting report on the analysis of higher-level languages is Shivers's approximation semantics for discovering the possible types of variables in Scheme programs [Shi90]. More study is needed to determine whether these techniques, which emphasize the control-flow-tracing aspect of program analysis, can be integrated with techniques for obtaining accurate estimates of store configurations discussed in Chapter 6.

One of the most difficult unsolved problems in program analysis is the development of efficient and effective techniques that support ***reference arithmetic***. One approach to handling pointer arithmetic in C, discussed by Allen and Johnson, uses heuristics to identify reference expressions that are used to step through arrays—and to replace these with operations on array indices [All88]. (*N.B.*: Allen and Johnson also mention earlier algorithms for induction variable elimination by Morel and Renvoise, and Chow; they state that these algorithms were not efficient enough for their purposes.) A second idea for handling pointer arithmetic, suggested by Ebcioglu, uses a combination of static analysis and dynamic reference checking to improve a program's behavior [*private communication*]. The static analysis would make the optimistic assumption that arithmetic operations on pointers can be replaced by equivalent, arithmetic-free operations that step through regular structures. The analysis would then detect those points in a program's execution at which this assumption might fail. Run-time checks inserted at points of possible failure would then be used to verify that assumptions about reference arithmetic are preserved at run-time. This proposal has much in common with recent work on combining static and dynamic type checking—discussed (*e.g.*) in a recent paper by Cartwright and Fagan [Car91].

A second important limitation of this thesis is the lack of attention given to ***alternative characterizations*** of program dependence. The algorithms developed in this thesis characterize the dependences that a program might exhibit, w.r.t. a standard interpretation of $\mathcal{H}$. As Section 3.4 points out, there are other important notions of dependence that these algorithms do not support. These algorithms, for example, do not identify the set of dependences that a pointer program ***must exhibit***. They also fail to capture a program's ***intended behavior*** (*cf.* §3.4.9). Both of these goals can probably be accomplished by replacing the abstract domain used in Chapters 5 and 6 with other, standard domains. More specifically, the interpretation developed in Chapters 5 and 6, which ignores states that characterize possible errors in a program evaluation, is modeled on a type of abstract domain known as a *lower* (or Hoare) powerdomain [Son87]. Standard techniques for modeling *intended* and *must* behavior use the *convex* (or Smyth-Plotkin) and *upper* (or Egli-Milner) powerdomains, respectively.

The ***pragmatic aspects*** of dependence computation have also been given short shrift. Chapter 6 observes that more research should be done on the relative ***performance*** of the various techniques for estimating stores. Chapter 6 also observes that the use of ***hybrid store estimation techniques***—including techniques that generate regular estimates of a program's stores—warrants further investigation.

Section 7.6 observes that the *hsdg*'s characterization of program behavior may be ***too complicated*** in practice. Section 7.6 also suggested possible fixes for the *hsdg*'s two principal problems: *i.e.*, its failure to encapsulate interprocedural dependences, and its reliance on def-order dependence. These fixes, however, are merely ideas for future research.

A final limitation of the thesis is its failure to demonstrate the ***soundness of other common transformations on dependence graphs***: *e.g.*, transformations that slice and splice *dbrs*, and transformations that use *dbrs* to parallelize a program's execution. The two principal reasons for not proving additional theorems about *hsdgs* are pragmatic.

- Since Chapter 7 is already long and involved, it seemed reasonable to devote a separate report to these concerns. The author firmly believes that the reduction developed in Chapter 7 can be used to show that other kinds of transformations on pointer-program *dbrs* are sound. This belief is based, in part, on the simplicity of the insight that underlies the proof of the Pointer-Language Equivalence Theorem: *i.e.*, the observation that assertions about terminating pointer programs can be reduced to comparable assertions about pointer-free languages.
- Making the effort to demonstrate additional theorems about *hsdgs* also seems inappropriate at this time. As Section 7.6 observes, the *hsdg* has important limitations that make it unattractive for certain types of program analysis. It seems reasonable to address this problem before attempting (*e.g.*) to extend Selke's calculus of *pdgs* to pointer-program *dbrs*.

This discussion of open problems raises a final question about the content of the thesis: why have foundational concerns been stressed at the expense of pragmatic ones? There are two answers to this question. The first is that the pointer-program analysis is a broad and complicated subject; the author had hoped to do more, but simply ran out of time. The second, more defensible answer is that the author believed that such concerns had not been given the attention they deserved. It is certainly important to develop new, more effective algorithms for estimating pointer-program behavior. It is also important, however, to make sure that these algorithms are correct—to ensure, in effect, that analyses are not simply generating "abstract nonsense". This thesis takes an important step in this direction: it shows that a family of algorithms yield provably safe estimates of a program's behavior, and that these estimates can—with certain caveats about freelists—be used to reason about program behavior. It is hoped that these results will provide a solid foundation for the work that must surely follow.

9. ACKNOWLEDGMENTS AND DEDICATIONS

> *I would like to acknowledge ... J.S. Bach, whose compositions often inspired my long hours of composing (semantic domains, equations, and predicates, that is); and whose music will still be remembered long after this dissertation has disappeared on some obscure microfilm.* —U. Pleban [Ple81]

When I first read Pleban's remark, it brought to mind the artists who helped *me* last through long sessions at the terminal: George Benson, Art Blakey, Dave Brubeck, Ron Carter, John Coltrane, Paul Desmond, Duke Ellington, Art Farmer, Erroll Garner, Benny Goodman, Dexter Gordon, Stephane Grappelli, David Grisman, Vince Guaraldi, Freddie Hubbard, Milt Jackson, Ahmad Jamal, Clifford Jordan, Jay McShann, Thelonius Monk, Gerry Mulligan, Makato Ozone, Charlie Parker, Art Pepper, Oscar Peterson, Bud Powell, Django Reinhardt, Wayne Shorter, Michael Urbaniak, and Teddy Wilson—and especially Bill Evans, whose piano music helped me through long nights at the terminal.

The remaining acknowledgements are more personal. The following individuals are thanked for their professional help and advice:

- my advisor, Thomas Reps, who impressed on me the importance of thoroughness, correctness, and attention to detail, and whose insistence on clarity made this a much more readable dissertation;
- Marvin Solomon, who read this thesis, and first brought to Tom's attention the Jones-Muchnick paper that led to this work;
- Charles Fischer, who also read this thesis, and first suggested the term "embedding";
- Richard Brualdi and Susan Horwitz, the other two members of my thesis committee;
- Lorenz Huelsberggen, who read and critiqued early drafts of this thesis;
- Robert Paige, who suggested that I investigate Schwartz's work;
- Rebecca Selke, without whose help Chapter 7 would not have been written;
- Thomas Murtagh, who took time to send me a copy of Ruggieri's thesis;
- David Chase, Patrick Cousot, Ron Cytron, Vince Guarna, Laurie Hendren, Flemming Nielson, Uwe Pleban, Olin Shivers, and Jan Stransky, who generously provided copies of their work;
- David Chase and Jan Stransky, who also took time to explain fine points of their work;
- Mark Wegman and Ken Zadeck, who shared with me their observations on pointer analysis;
- Maya Gokhale, Lauren Smith, and the people at SRC, who invited me to speak about these results;
- G.A. Venkatesh, whose helpful observations on semantics finally started to make sense to me this past year, years after he shared them with me;
- David Binkley and Wuu Yang, who were always ready to discuss their research; and
- The UW computer lab staff, who were quick to answer my many questions about *troff* and *idraw*.

The following people, who are connected with the UW—Madison, are thanked for showing me various kindnesses:

- Professors Jim Larus and Jim Goodman, Tom Ball, Sam Bates, Dave Binkley, Tom Bricker, Bob Holloway, Lorenz Huelsberggen, Robert Netzer, and Todd Proebsting, who lent emotional support and encouragment;
- many others—including Mark Allmen, George Bier, Edie Epstein, Mike Franklin, Lou Goodman, Wei Hsu, Rick Kessler, Sanjay Krishnamurthy, Tripp Lazarus, Gary Lewandoski, Dan Lieuwen, Olvi Mangasarian, Amarnath Mukherjee, G. Ramalingam, Tony Rich, Cheng Song, Jon Sorenson,

Divesh Srivastava, S. Sudharshan, Peter Sweeney, Phil Woest, and Wuu Yang—for simply being the sort of people who smiled and said "hello" when passing through the halls;

- Verallyn Cline and Bonnie Griswold, for being two of the best instructors and friends that I had while at the UW; and
- the department's secretarial staff—especially Laura Cuccia, Lorene Webber, and Lynn White—for their warmth, efficiency, and friendliness.

I also wish to acknowledge the support of the following people, who are unconnected with the UW—Madison:

- My wife, Linda, who encouraged me to return to graduate school at the advanced advanced age of 30, and without whose support I could not have completed;
- my family, especially my parents and my grandmother, who have seen, experienced, and suffered through so much more than I; and
- the out-of-town friends who made the much-appreciated effort to visit us in Madison: Marilyn Feldhaus and David Gross, Anne Harnack, Debbie Heylmun, Wei Hsu and Diana Li, Jim McDonald, Michael and Cindy Loui, David Sherman, and (especially) Chuck and Mary Ellen Netzel.

Finally, I think it appropriate to dedicate this thesis to the late Alan Perlis, who taught my first computer course back in 1972, and the late David Kamowitz (Ph.D., UW—Madison, 1986), the news of whose death came as one of saddest surprises of my seven years in grad school.

Until it ends, there is no end.—Cyndi Lauper

But then it's all over.—Charles Fischer

Appendix 1. A Semantics for Language $\mathcal{H}$

The following is a formal semantics for the language $\mathcal{H}$. Additional remarks follow the semantics.

State	$= Point \times Store \times Freelist$	*Freelist*	$= Loc^*$
Store	$= Loc \rightarrow Struct$	*StructDeclEnv*	$= Type \rightarrow pwr(Sel)_\perp$
Struct	$= Type \times Atom \times Context \times Selmap$	*Context*	$= Point$
Selmap	$= Sel \rightarrow Loc$		

The expression $pwr(D)$, where D is a set, denotes the powerset of D.

$\mathbf{M}_{\mathcal{H}}: Prog \rightarrow Store \rightarrow Store_\perp$
$\mathbf{M}_{\mathcal{H}}(prog, \sigma) =$
let $(structdecls, body) = prog$ **in**
let $body' = expand(initialize(body))$
and $structDecls = evalStructDecls(structdecls)$
and fl = an infinite, nonrepeating list of locations not in σ
in
let $evalPgm = fix\ \lambda f.\ \lambda((pt, \sigma', fl)).\ pt \stackrel{?}{=} \mathbf{final} \rightarrow \sigma'\ []\ f(evalPt((pt, \sigma', fl)))$
in $evalPgm((\mathbf{initial}_1, \sigma, fl))$
end*

Function *fix* is the least fixpoint functional.

The function *initialize* (*text*) appends the following three-line statement list to *text*:

$[\mathbf{initial}_1]$ *initialize* ; $[\mathbf{initial}_2]$ *call main*() ; $[\mathbf{final}]$ *skip* ;

The function *expand* (*text*) replaces *text* with a related program text that "materializes" *text*'s points of control:

- Every statement in *text* of the form $[p]$ **call** $A(a_1, \cdots, a_n)$ is replaced by the following statements:

```
[p.i0]   _curr._δ0 := saveContext [nextp];   /*** pass return point to callee ***/
                                             /*** [nextp] is point p's control-flow successor ***/
[p.i1]   _curr._δ1 := a1;                    /*** pass actual parameters to callee ***/
...
[p.in]   _curr._δn := an;
[p.c]    call A;                             /*** perform the call ***/
```

- The following sequence of statements is placed at the head of a procedure A with formals $f_1 \cdots f_n$:

```
[A.i-3]  _temp := new(env);                  /*** initialize new local environment ***/
[A.i-2]  _temp._prev := _curr;
[A.i-1]  _curr := _temp;
[A.i0]   _curr._callctxt := _curr._prev._δ0; /*** get return point from caller ***/
[A.i1]   _curr.f1 := _curr._prev._δ1;        /*** get values of formals from caller ***/
...
[A.in]   _curr.fn := _curr._prev._δn;
```

- The statement "[*A.f*] *return*" is placed immediately after the final executable statement in the body of A.

```
/* *** *** declarations *** *** */

evalStructDecls : DeclList → StructDeclEnv
evalStructDecls(decllist) =
 let evalList =
     fix λ f . λ (declList, declEnv) .
       declList ≟ ε → declEnv
        [] let (decl, declList') = declList in
             let (type, (sel_1, ···, sel_n)) = decl
             in f(declList', declEnv[{ sel_1, ···, sel_n } / type])
           end•
 in
   let progDecls = evalList(declList, λ type . ∅) in progDecls [ ∅ / atom ] [ ∅ / context ]
 end•

/* *** program points *** */

evalPt : State → State_⊥
evalPt ((pt, σ, fl)) =
 case formOf (pt, body') in
   If (cexp) , While (cexp):  let nextpt_T and nextpt_F be pt's true and false control-flow successors
                              in (cond (pt, σ, cexp) → nextpt_T [] nextpt_F, σ, fl)
                              end
   Assign (lexp, rexp):       let nextpt be pt 's control-flow successor in
                               let (σ', fl', src) = simplexp (pt, σ, fl, rexp) in
                                 let tgt = idexpr (pt, σ', lexp) in
                                   let (σ'', fl'', tgt') =
                                     ((gettype (σ', tgt) ≟ atom → addatom(σ', fl', getval (σ', tgt)) [] (σ', fl', tgt))
                                    in (nextpt, updref (σ'', src, sel, tgt'), fl'')
                              end•
   Call (proc):               let nextpt be proc's entry point in (nextpt, σ, fl) end
   Return ( ):                let retpt = getcxt(σ, idexpr(pt, σ, _curr._callctxt))
                              and prevenv = idexpr (pt, σ, _curr._prev)
                              and globalEnv be σ's global environment in
                               let σ' = updref (σ'', globalEnv, _curr, prevenv)
                               in (retpt, σ', fl)
                              end•
   Initialize ( ):            let nextpt be pt 's control-flow successor
                              and σ' be a copy of σ
                              in
                               let globalEnv be σ''s global environment in
                                 let σ'' = newref (σ, globalEnv, _curr, globalEnv)
                                 in (nextpt, σ'', fl)
                              end•
 esac
```

Function *formOf* pairs every program point with its associated syntactic construct.

```
/* *** *** expressions *** *** */

cond : Point × Store × Cond → Bool⊥
cond (pt, σ, cexp) =
 case cexp in
   TypeOf (exp,type):      gettype(σ, idexpr (pt, σ, exp)) ?= type
   Eq (exp1, exp2):        idexpr (pt, σ, exp1) ?= idexpr (pt, σ, exp2)
   Compare (exp1, op, exp2): let loc1 = idexpr (pt, σ, exp1) and loc2 = idexpr (pt, σ, exp2) in
                             { gettype (σ, loc1), gettype (σ, loc2) } ≠ { atom } → ⊥
                             [] getval(σ, loc1)  op  getval(σ, loc2)
                           end
   Not (cexp):             ¬ cond (pt, σ, cexp)
 end

simplexp : Point × Store × Freelist × Exp → (Store × Freelist × Loc)⊥
simplexp (pt, σ, fl, exp) =
 case exp in
   Selexp (sexp):    (σ, fl, idexpr (pt, σ, sexp))
   Atom (a):         addatom (σ, fl, a)
   SaveContext (p):  addcontext (σ, fl, p)
   New (type):       let (σ1, fl1, loc) = addstruct (σ, fl, type) in
                      if type ?= env then (σ1, fl1, loc)
                      else
                        let initfields = fix λ f . λ (σ', fl', selset).
                          selset ?= ∅ → (σ', fl', loc)
                          [] let (σ'', fl'', at) = addatom (σ', fl', nil) in
                               let sel be an element of selset in
                                 let σ''' = newref (σ'', loc, sel, at)
                                 in f(σ''', fl'', selset – { sel })
                             end*
                        in initfields(σ1, fl1, structDecls (type))
                        end
                      fi
                     end
   Primop (op, exp1, ···, expn):
                     let (σ1, fl1, loc1) = simplexp (pt, σ, fl, exp1) in
                       ···
                       let (σn, fln, locn) = simplexp (pt, σn-1, fln-1, expn)
                       in addatom (σn, fln, op(loc1, ···, locn))
                     end*
 esac

/* *** *** primitive operations on stores *** *** */

updref: Store × Loc × Sel × Loc → Store⊥
updref(σ, src, sel, tgt) =
 ( gettype (σ, src) ?= env ∨ sel ∈ structDecls (gettype (σ, src)) ) → newref(σ, src, sel, tgt) [] ⊥
```

newref : $Store \times Loc \times Sel \times Loc \rightarrow Store_{\perp}$
newref $(\sigma, src, sel, tgt) =$ **let** $(typ, val, cxt, map) = \sigma(src)$ **in** $\sigma[\ (typ, val, cxt, map\ [tgt\ /\ sel])\ /\ src\ ]$ **end**

addatom : $Store \times Freelist \times Atom \rightarrow Store \times Freelist \times Loc$
addatom $(\sigma, fl, val) =$ **let** $(fl', loc) = alloc(fl)$ **in** $(\sigma[(\textbf{atom}, val, \perp, \perp)\ /\ loc], fl', loc)$ **end**

addcontext : $Store \times Freelist \times Point \rightarrow Store \times Freelist \times Loc$
addcontext $(\sigma, fl, pt) =$ **let** $(fl', loc) = alloc(fl)$ **in** $(\sigma[(\textbf{context}, \perp, pt, \perp)\ /\ loc], fl', loc)$ **end**

addstruct : $Store \times Freelist \times Type \rightarrow Store \times Freelist \times Loc$
addstruct $(\sigma, fl, typ) =$ **let** $(fl', loc) = alloc(fl)$ **in** $(\sigma[(typ, \perp, \perp, \perp)\ /\ loc], fl', loc)$ **end**

idexpr : $Point \times Store \times Idexp \rightarrow Loc_{\perp}$
idexpr $(pt, \sigma, id.sexp) =$
let *globalEnv* be σ's global environment
in *selexp* $(\sigma, globalEnv, (\ id \in localIds(pt, body') \rightarrow \_curr.id.sexp\ []\ id.sexp\ ))$
end

localIds returns the set of variables that are local at point *pt*.

selexp : $Store \times Loc \times (Ident + Sel)^* \rightarrow Loc_{\perp}$
selexp $(\sigma, loc, sexp) =$
$sexp \stackrel{?}{=} \varepsilon \rightarrow loc$ [] **let** $(sel, sexp') = sexp$ **and** $map = getmap(\sigma, loc)$ **in** $selexp(\sigma, map(sel), sexp')$ **end**

gettype : $Store \times Loc \rightarrow Type =$ $\lambda(\sigma, loc)$. **let** $(typ, \_, \_, \_) = \sigma(loc)$ **in** *typ* **end**

getval : $Store \times Loc \rightarrow Atom =$ $\lambda(\sigma, loc)$. **let** $(\_, val, \_, \_) = \sigma(loc)$ **in** *val* **end**

getcxt : $Store \times Loc \rightarrow Context =$ $\lambda(\sigma, loc)$. **let** $(\_, \_, cxt, \_) = \sigma(loc)$ **in** *cxt* **end**

getmap : $Store \times Loc \rightarrow Selmap =$ $\lambda(\sigma, loc)$. **let** $(\_, \_, \_, map) = \sigma(loc)$ **in** *map* **end**

alloc : $Freelist \rightarrow Loc \times Freelist = \lambda fl$. **let** $(loc, fl') = fl$ **in** (loc, fl') **end**

To simplify the semantics, the following variables are treated as global objects:

* *body'* (defined by $\mathbf{M}_{\mathcal{H}}$ and manipulated in *evalPt* and *idexpr*);
* *structDecls* (defined by $\mathbf{M}_{\mathcal{H}}$ and manipulated by *updref*);

A *program point* is a unique name associated with each of a program's assignment statements, *if* and *while* predicates, *call*, and *skip* statements. Program points, which are members of domain *Point*, are used to monitor a program's evaluation. Points **initial**$_1$, **initial**$_2$, and **final** are special points that correspond to steps in a program's initialization termination routines, respectively. Other special points are associated with procedure call and return.

Domain *Loc* is a domain of objects that "contain" structures. To simplify the semantics, it is assumed that any $loc \in Loc$ can contain any type of structure.

Function $\mathbf{M}_{\mathcal{H}}$ is the program meaning function. Every initial store σ passed to $\mathbf{M}_{\mathcal{H}}$ must meet the four requirements for initial stores given in Chapter 2.

Function *evalStructDecls* processes a program's structure declarations. It generates a function, *structDecls*, that names, for every user-defined type *struct*, those selectors that *struct* accepts. Function

evalStructDecl also initializes two built-in types. These types, **context** and **atom**, accept no selectors.

Specially named references in the caller's environment are used to pass parameters to callees. Reference $\_\delta_0$ is reserved for the return context. References $\_\delta_1 \cdots \_\delta_k$ are reserved for the first through kth parameters of subroutine calls, respectively. The assumption that a program point is stored and accessed like any other atom is another simplifying assumption.

The *Initialize* case of *evalPt* handles program initialization. The evaluation of the initial program point (*i.e.*, the clause "**let** $\sigma' = copy(\sigma)$") creates every object in the program's store as that program begins its evaluation.

Function *cond* evaluates conditional expressions. To simplify the definition of *cond*, no distinction is made between the symbol for a comparison operation and the operation *per se*. Also, the domain of atoms is assumed to be totally ordered by the relational operators "<" and ">".

Function *simplexp* evaluates expressions. The following comments apply to *simplexp*:

* No distinction is made, in *Primop* case of *simplexp*, between the symbol for an operator and the operator *per se*.
* The *New* case of *simplexp* initializes a non-environment structure *s* by linking each of *s*'s fields to new, nil-valued atoms.

Functions *addatom* and *addstruct* add new atoms and structures to the store. Function *newref* adds new references to the store. Function *updref* updates existing references in the store. Note that *updref* checks that the structure named by its *loc* argument accepts the structure named by its *sel* argument.

Functions *idexpr* and *selexp* interpret the meaning of selector expressions.

The *alloc*( ) function removes the first location from the freelist, and returns this location to the caller. The definition of *alloc*( ) assumes that the freelist is inexhaustible.

Appendix 2. An Instrumented Semantics for Language $\mathcal{H}$

This appendix describes an instrumented semantics for the language $\mathcal{H}$, function $\mathbf{MI}_{\mathcal{H}}$. $\mathbf{MI}_{\mathcal{H}}$ differs from $\mathbf{M}_{\mathcal{H}}$ in the following two regards. $\mathbf{MI}_{\mathcal{H}}$ maintains a *label* function that pairs every non-atomic object o with the occurrence of the point that created o. $\mathbf{MI}_{\mathcal{H}}$ also maintains a computation's occurrence string.

Semantics $\mathbf{MI}_{\mathcal{H}}$ has four altered and four new domains:

$$
\begin{array}{llllll}
State_I & = Point \times Store_I \times Freelist \times Occ \times Label & & & & \\
Store_I & = Loc \rightarrow Struct_I & & & & \\
Struct_I & = Type \times Atom \times Context_I \times Selmap & Context_I & = Point \times Occ & Occ & = Point^* \\
Label & = Loc \rightarrow StructLabel \times RefLabel & StructLabel & = Occ & RefLabel & = Sel \rightarrow Occ
\end{array}
$$

Comments on specific differences between $\mathbf{M}_{\mathcal{H}}$ and $\mathbf{MI}_{\mathcal{H}}$ follow the semantics.

$\mathbf{MI}_{\mathcal{H}}: Prog \rightarrow Store_I \rightarrow (Store_I)_\perp$

$\mathbf{MI}_{\mathcal{H}}(prog, \sigma) =$

let $(structdecls, body) = prog$ **in**

let $body' = expand(initialize(body))$

and $structDecls = evalStructDecls(structdecls)$

in

let fl = an infinite, nonrepeating list of locations not in σ

and $label = \lambda\, loc\,.\,(undefined, \lambda\, sel\,.\, undefined)$

and $occ = \varepsilon$, the empty occurrence string

in

let $evalPgm_I = fix\ \ \lambda\, f\,.\,\lambda\,((pt, \sigma', fl', label', occ'))\,.\ pt \stackrel{?}{=} \mathbf{final} \rightarrow \sigma'\ []\ f(evalPt_I\,((pt, \sigma', fl', label', occ')))$

in $evalPgm_I\,((\mathbf{initial}_1, \sigma, fl, label, occ))$

end*

The definitions of *initialize*, *expand*, and *evalStructDecls* are unchanged from Chapter 2.

```
/* *** *** program points *** *** */

evalPt_I : State_I → State_I⊥

evalPt_I (state) =
 let (pt, σ, fl, label, occ) = state and ptocc = append(occ, pt) in
  case formOf (pt, body') in

    If (cexp):          let nextpt_T and nextpt_F be pt's true and false control-flow successors
                        in ((cond (pt, σ, cexp) → nextpt_T [] nextpt_F), σ, fl, label, occ)
                        end

    While (cexp):       let nextpt_T and nextpt_F be pt's true and false control-flow successors in
                         let nextocc_T = ptocc and nextocc_F = removeSuffix (occ, pt)
                         and bool = cond (pt, σ, cexp)
                         in ((bool → nextpt_T [] nextpt_F), σ, fl, label, (bool → nextocc_T [] nextocc_F))
                        end*

    Assign (lexp, rexp): let nextpt be pt's control-flow successor in
                         let (σ', fl', label', loc) = simplexp (state, rexp) in
                          let tgt = idexpr (pt, σ', lexp) in
                           let (σ'', fl'', tgt') =
                            ((gettype (σ', tgt) ≟ atom → addatom(σ', fl', getval (σ', tgt)) [] (σ', fl', tgt))
                           in
                            let (σ''', label'') = updref (σ'', label', ptocc, src, sel, tgt)
                            in (nextpt, σ''', fl'', label'', occ)
                        end*

    Call (proc):        let nextpt be proc 's entry point in (nextpt, σ, fl, label, ptocc) end

    Return ( ):         let (retpt, retocc) = getcxt (σ, idexpr (pt, σ, _curr._callctxt))
                        and prevenv = idexpr (pt, σ, _curr._prev)
                        and globalEnv be σ's global environment in
                         let (σ', label' ) = updref (σ, label, ptocc, globalEnv, _curr, prevenv)
                         in (retpt, σ', fl, label', retocc)
                        end*

    Initialize ( ):     let nextpt be pt's control-flow successor
                        and σ' be a copy of σ
                        and label' be the function that pairs every accessible non-atomic object in σ with ptocc
                        in
                         let globalEnv be σ''s global environment in
                          let (σ'', label'') = newref (σ', label', ptocc, globalEnv, _curr, globalEnv)
                          in (nextpt, σ'', fl, label'', occ')
                        end*

  esac

 end
```

Function *formOf* pairs every program point with its associated syntactic construct.

The expression *removeSuffix* (*occ*, *pt*) denotes the empty occurrence string iff *occ* is of the form pt^k for some nonnegative k. Otherwise, let *occ* be of the form $p_1 \cdots p_j pt^k$, where $pt \neq p_j$, for some nonnegative k; then *removeSuffix* $(occ, pt) = p_1 \cdots p_j$.

```
/* *** *** expressions *** *** */
```

The definition of *cond* is the same as in Chapter 2.

```
simplexp : State_I × Exp → (Store_I × Freelist × Label × Loc)_⊥
simplexp (state, exp) =
 let (pt, σ, fl, label, occ) = state and ptocc = append(occ, pt) in
  case exp in
    Selexp (sexp):    (σ, fl, label, idexpr (pt, σ, sexp))
    Atom (a):         let (σ', fl', loc) = addatom (σ, fl, a) in (σ', fl', label, loc) end
    SaveContext (p):  addcontext (σ, fl, label, ptocc, p, occ)
    New (type):       let (σ_1, fl_1, label_1, loc) = addstruct (σ, fl, label, ptocc, type) in
                       if type ?= env then (σ_1, fl_1, label_1, loc)
                       else
                         let initfields =fix λ f . λ (σ', fl', label', selset) .
                           selset ?= ∅ → (σ', fl', label', loc)
                           [] let (σ'', fl'', at) = addatom (σ', fl', nil) in
                              let sel be an element of selset in
                                let (σ''', label'') = newref (σ'', label'', ptocc, loc, sel, at)
                                in f (σ''', fl'', label'', selset – { sel })
                              end*
                         in initfields(σ, fl, label, structDecls (type))
                         end
                       fi
                      end
    Primop (op, exp_1, ··· , exp_n):
                      let ((σ_1, fl_1, label_1), loc_1) = simplexp (state, exp_1) in
                        ···
                        let ((σ_n, fl_n, label_n), loc_n) = simplexp ((pt, σ_{n-1}, fl_{n-1}, label_{n-1}, occ), exp_n) in
                          let (σ', fl', loc) = addatom (σ_n, fl_n, op(loc_1, ··· , loc_n))
                          in (σ', fl', label_n, loc)
                        end*
  esac
 end

/* *** *** primitive operations on stores *** *** */

updref : Store_I × Label × Occ × Loc × Sel × Loc → (Store_I × Label)_⊥
updref (σ, label, ptocc, src, sel, tgt) =
 if (gettype (σ, src) ?= env ∨ sel ∈ structDecls (gettype (σ, src))) then newref(σ', label', ptocc, src, sel, tgt)
 else ⊥
 fi

newref : Store_I × Label × Occ × Loc × Sel × Loc → (Store_I × Label)_⊥
newref (σ, label, ptocc, src, sel, tgt) =
 let (type, val, cxt, map) = σ(src) and (srcdef, refdefs) = label(src) in
  let newstruct = (type, val, cxt, map [tgt / sel]) and newlbl = (srcdef, refdefs[ptocc / sel])
  in (σ[ newstruct / src ], label [ newlbl / src ])
 end*
```

$addstruct : Store_I \times Freelist \times Label \times Occ \times Type \rightarrow Store_I \times Freelist \times Label \times Loc$
$addstruct\ (\sigma, fl, label, ptocc, type)$
let $(fl', loc) = alloc\ (fl)$ **in**
let $newstruct = (type, \perp, bottom, \perp)$ **and** $newlbl = (ptocc, \lambda\ sel\ .\ \varnothing)$
in $(\sigma[\ newstruct\ /\ loc\ ], fl', label[\ newlbl\ /\ loc\ ], loc)$
end•

$addcontext : Store_I \times Freelist \times Label \times Occ_A \times Point \times Occ \rightarrow Store_I \times Freelist \times Loc$
$addcontext\ (\sigma, fl, label, ptocc, pt, occ)$
let $(fl', loc) = alloc\ (fl)$ **in**
let $newstruct = (\textbf{context}, \perp, (pt, occ), \perp)$ **and** $newlbl = (ptocc, \lambda\ sel\ .\ \varnothing)$
in $(\sigma[\ newstruct\ /\ loc\ ], fl', label[\ newlbl\ /\ loc\ ], loc)$
end•

The definitions of *addatom*, *idexpr*, *selexp*, *gettype*, *getval*, *getcxt*, and *getmap* are similar to the ones given in Chapter 2. The instrumented semantics' versions of these functions accepts a member of $Store_I$ as a parameter.

The definition of *alloc* is unchanged from Chapter 2.

$\mathbf{MI}_{\mathcal{H}}$ proper differs from $\mathbf{M}_{\mathcal{H}}$ proper in two ways. $\mathbf{MI}_{\mathcal{H}}$ passes an initial label and occurrence string to $evalPgm_I$. $evalPgm_I$ then passes the current label and occurrence string to $evalPt_I$.

$evalPt_I$ differs from $\mathbf{M}_{\mathcal{H}}$'s version of $evalPt_I$ in the following ways:

- $evalPt_I$ accepts, and returns, instrumented states.
- The interpretation of the while statement updates a computation's occurrence string.
- The interpretation of the *return* statement restores the previous occurrence string.
- The interpretation of the initialization statement affixes creation labels to newly created objects.
- Labeling information is passed to *simplexp* and *updref*.

simplexp differs from $\mathbf{M}_{\mathcal{H}}$'s version of *simplexp* in the following ways:

- *simplexp* accepts an instrumented state, and returns an updated *label* parameter.
- The interpretation of the *saveContext* operator saves the current occurrence string.
- Labeling information is passed to *addstruct* and *newref*.

The instrumented semantics' versions of *updref*, *newref*, and *addstruct* update program point labels on newly defined objects.

Appendix 3. An Approximation Semantics for Language $\mathcal{H}$

The following is an instrumented approximation semantics for language $\mathcal{H}$. This semantics is discussed in Section 5.1.

$State_A = Point \times Store_A \times Freelist \times Label_A \times Occ_A$

$Store_A = Loc \rightarrow Struct_A$

$Struct_A = Kind \times pwr(Type) \times pwr(Atom) \times Context_A \times Selmap_A$

$Selmap_A = Sel \times Int \rightarrow Loc$

$Label_A = Loc \rightarrow RefLabel_A \times SelLabel_A$

$RefLabel_A = pwr(Occ_A)$

$SelLabel_A = Sel \times Int \rightarrow pwr(Occ_A)$

Occ_A is the domain of program-point regular expressions

$\mathbf{MA}_{\mathcal{H}}: Store_A \rightarrow pwr(Store_A)$

$\mathbf{MA}_{\mathcal{H}}(prog, \sigma) =$

let $(structdecls, body) = prog$ **in**
  let $body' = expand(initialize(body))$
  and $structDecls = evalStructDecls(structdecls)$
  in
    let fl = an infinite, nonrepeating list of locations not in σ
    and $label = \lambda\, loc\,.\,(\{\, undefined \,\}, \lambda\, sel\,.\,\{\, undefined \,\})$
    in
      let $evalPgm_A = fix\ \lambda f.\lambda(curr_\bullet, iterct).$
        let $next_\bullet =$ **union_from** $state \in curr_\bullet :\ evalPt_A(state)$ **in**
          let $next'_\bullet = \nabla(iterct, next_\bullet)$ **in**
            $next'_\bullet \sqsubseteq curr_\bullet \rightarrow curr_\bullet\ []\ f(next'_\bullet, iterct+1)$
        end*
      in $evalPgm_A(\{\,(\mathbf{initial}_1, \sigma, fl, label, \varepsilon)\,\}, 0)$
end*

The definitions of *evalStructDecls*, *initialize*, and *expand* are the same as in Chapter 2.

The $\sqsubseteq$ relation identifies pairs of states that are redundant from the standpoint of dependence computation. A definition of $\sqsubseteq$ is given in Appendix 4.

The operator ∇ has the signature $Nat \times pwr(State_A) \rightarrow pwr(State_A)$. It is assumed that ∇ is *extensive* w.r.t. $\sqsubseteq$: *i.e.*, that $next_\bullet \sqsubseteq \nabla(i, next_\bullet)$ for all $i \in Nat$ and all $next_\bullet \in pwr(State_A)$. It is also assumed that ∇ ensures that P terminates on σ: *e.g.*, by limiting the number of structures in stores, or by limiting the number of steps in an analysis. Possible definitions for ∇ are given in Chapter 6.

$evalPt_A : State_A \rightarrow pwr(State_A)$
$evalPt_A\,(state) =$
let $(pt, \sigma, fl, label, occ) = stmt$ **and** $ptocc = append(occ, pt)$ **in**
case $formOf\,(pt, body')$ **in**

If (*cexp*):
let $nextpt_T$ and $nextpt_F$ be pt 's true and false control-flow successors **in**
let $states_{*T} = cond_T\,(pt, \sigma, cexp) \rightarrow \{\,(nextpt_T, \sigma', fl, label, occ)\,\}$ [] $\varnothing$
and $states_{*F} = cond_F\,(pt, \sigma, cexp) \rightarrow \{\,(nextpt_F, \sigma', fl, label, occ)\,\}$ [] $\varnothing$
in $states_{*T} \cup states_{*F}$
end•

While (*cexp*):
let $nextpt_T$ and $nextpt_F$ be pt 's true and false control-flow successors
and $nextocc_T = ptocc$ **and** $nextocc_F = removeSuffix(occ, pt)$
in
let $states_{*T} = cond_T\,(pt, \sigma, cexp) \rightarrow \{\,(nextpt_T, \sigma', fl, label, nextocc_T)\,\}$ [] $\varnothing$
and $states_{*F} = cond_F\,(pt, \sigma, cexp) \rightarrow \{\,(nextpt_F, \sigma', fl, label, nextocc_F)\,\}$ [] $\varnothing$
in $states_{*T} \cup states_{*F}$
end•

Assign (*lexp.sel, rexp*):
let $nextpt$ be pt 's control-flow successor **in**
union_from $(\sigma', fl', label', rvloc) \in simplexp_A\,(pt, state, rexp)$:
union_from $lvloc \in idexpr_A\,(pt, \sigma', lexp)$:
union_from $(\sigma'', fl'', label'') \in updref(\sigma', fl', label', ptocc, lvloc, sel, rvloc)$:
$\{\,(nextpt, \sigma'', fl'', label'', occ)\,\}$
end

Call (*proc*): **let** $nextpt$ be $proc$'s entry point **in** $\{\,(nextpt, \sigma, fl, label, ptocc)\,\}$ **end**

Return ( ):
union_from $cxtloc \in idexpr_A\,(pt, \sigma, \_curr.\_callctxt)$:
union_from $(retpt, apxocc) \in getcxt_A\,(\sigma, cxtloc)$:
union_from $prevenv \in idexpr_A\,(pt, \sigma, \_prev)$ such that *consistent* $(retpt, label, prevenv)$:
let $globalEnv$ be σ''s global environment **in**
let $(\sigma', label') = updref\,(\sigma, label, ptocc, globalEnv, \_curr, prevenv)$
in $\{\,(retpt, \sigma', fl, label', apxocc)\,\}$
end•

Initialize ( ):
let $nextpt$ be pt's control-flow successor
and σ' be a copy of σ
and $label'$ be the function that pairs every accessible non-atomic object in σ with $\{\,ptocc\,\}$
in
let $globalEnv$ be σ''s global environment **in**
let $(\sigma'', label'') = newref\,(\sigma', label', ptocc, globalEnv, \_curr, globalEnv)$
in $\{\,(nextpt, \sigma', fl, label', occ')\,\}$
end•

esac
end

Function *formOf* pairs every program point with its associated syntactic construct.

The predicate *consistent* (*retpt, label, prevenv*) is true iff *retpt* is a point in procedure *P* and *prevenv*'s creation-point label implies that *prevenv* might have been created by *P*.

/* *** *** expressions *** *** */

$cond_T : Point \times Store_A \times Cond \rightarrow Bool$

$cond_T\,(pt,\ \sigma,\ cexp) =$

case *cexp* **in**

TypeOf $(exp, type)$: $\exists\ loc \in idexpr_A\,(pt,\ \sigma,\ exp)$: *maybe_oftype*$(\sigma,\ loc,\ type)$

Eq $(exp_1,\ exp_2)$:
let *may_coincide* $= \lambda\,(loc',\ loc'').\ \ loc' \stackrel{?}{=} loc''$
in $\exists\ loc_1 \in idexpr_A\,(pt,\ \sigma,\ exp_1) : \exists\ loc_2 \in idexpr_A\,(pt,\ \sigma,\ exp_2)$: *may_coincide*$(loc_1,\ loc_2)$
end

Compare $(exp_1,\ op,\ exp_2)$:
let *may_satisfy* $= \lambda\,(loc',\ op',\ loc'')$. **true** $\in (getval_A\,(\sigma,\ loc')\ \ op'\ \ getval_A\,(\sigma,\ loc''))$
in
$\exists\ loc_1 \in idexpr_A\,(pt,\ \sigma,\ exp_1) : \exists\ loc_2 \in idexpr_A\,(pt,\ \sigma,\ exp_2)$:
maybe_atom$(\sigma,\ loc_1) \wedge$ *maybe_atom*$(\sigma,\ loc_2) \wedge$ *may_satisfy*$(loc_1,\ op,\ loc_2)$
end

Not $(cexp)$: $\quad cond_F\,(pt,\ \sigma,\ cexp)$

esac

$cond_F : Point \times Store_A \times Cond \rightarrow Bool$

$cond_F\,(pt,\ \sigma,\ cexp) =$

case *cexp* **in**

TypeOf $(exp, type)$: $\exists\ loc \in idexpr_A\,(pt,\ \sigma,\ exp)$: *maybe_not_oftype*$(\sigma,\ loc,\ type)$

Eq $(exp_1,\ exp_2)$:
let *may_differ* $= \lambda\,(\sigma',\ loc',\ loc'').\ \ loc' \neq loc'' \vee (loc' \stackrel{?}{=} loc'' \wedge$ *is_summary*$(\sigma',\ loc'))$
in $\exists\ loc_1 \in idexpr_A\,(pt,\ \sigma,\ exp_1) : \exists\ loc_2 \in idexpr_A\,(pt,\ \sigma,\ exp_2)$: *may_differ*$(\sigma,\ loc_1,\ loc_2)$
end

Compare $(exp_1,\ op,\ exp_2)$:
let *may_not_satisfy* $= \lambda\,(loc',\ op',\ loc'')$. **false** $\in (getval_A\,(\sigma,\ loc')\ \ op'\ \ getval_A\,(\sigma,\ loc''))$
in
$\exists\ loc_1 \in idexpr_A\,(pt,\ \sigma,\ exp_1) : \exists\ loc_2 \in idexpr_A\,(pt,\ \sigma,\ exp_2)$:
maybe_atom$(\sigma,\ loc_1) \wedge$ *maybe_atom*$(\sigma,\ loc_2) \wedge$ *may_not_satisfy*$(loc_1,\ op,\ loc_2)$
end

Not $(cexp)$: $\quad cond_T\,(pt,\ \sigma,\ cexp)$

esac

$simplexp_A : State_A \times Exp \rightarrow pwr(Store_A \times Freelist \times Label_A \times Loc)$
$simplexp_A\ (state, exp) =$
let $(pt, \sigma, fl, label, occ) = state$ **and** $ptocc = append(occ, pt)$ **in**
case *exp* **in**
Selexp (*sexp*): **union_from** $loc \in idexpr_A\ (pt, \sigma, sexp)$: { $(\sigma, fl, label, loc)$ }
Atom (*a*): **let** $(\sigma', fl', loc) = addatom_A\ (\sigma, fl, a)$ **in** { $(\sigma', fl', label, loc)$ }
SaveContext (*p*): { $addcontext_A\ (\sigma, fl, label, ptocc, p, occ)$ }
New (*type*): **let** $(\sigma_1, fl_1, label_1, loc) = addstruct_A\ (\sigma, fl, label, ptocc, type)$ **in**
if *type* $\stackrel{?}{=}$ **env** **then** { $(\sigma_1, fl_1, label_1, loc)$ }
else
let *initfields* = *fix* $\lambda f\ .\ \lambda\ (\sigma', fl', label', selset)$.
selset $\stackrel{?}{=} \emptyset \rightarrow$ { $(\sigma', fl', label', loc)$ }
[] **let** $(\sigma'', fl'', at) = addatom_A\ (\sigma', fl', \mathbf{nil})$ **and** *sel* be an element of *selset* **in**
let $(\sigma''', label'') = newref\ (\sigma'', label', ptocc, loc, sel, at)$
in $f(\sigma''', fl'', label'', selset - \{ sel \})$
end
in $initfields(\sigma_1, fl_1, label_1, structDecls(type))$
fi
end
Primop ($op, exp_1, \cdots, exp_n$):
union_from $(\sigma_1, fl_1, label_1, loc_1) \in simplexp_A\ ((pt, \sigma, fl, label, occ), exp_1)$:
$\cdots$
union_from $(\sigma_n, fl_n, label_n, loc_n) \in simplexp_A\ ((pt, \sigma_{n-1}, fl_{n-1}, label_{n-1}, occ), exp_n)$:
let $(\sigma', fl', loc') = addatom_A\ (\sigma_n, fl_n, op(loc_1, \cdots, loc_n))$
in { $(\sigma', fl', label, loc')$ }
end*
esac
end

/* *** primitive operations on stores *** */

$addatom_A : Store_A \times Freelist \times Atom \rightarrow Store_A \times Freelist \times Loc$
$addatom_A\ (\sigma, fl, val) =$ **let** $(fl', loc) = alloc\ (fl)$ **in** $(\sigma[(\mathbf{ordinary}, type, val, \perp, \perp)\ /\ loc], fl', loc)$ **end**

$addstruct_A : Store_A \times Freelist \times Label_A \times Occ_A \times Type \rightarrow Store_A \times Freelist \times Label_A \times Loc$
$addstruct_A\ (\sigma, fl, label, ptocc, type) =$
let $(fl', loc) = alloc\ (fl)$ **in**
let $struct' = (\mathbf{ordinary}, type, \perp, \perp, \perp)$ **and** $structlbl' = (\{ ptocc \}, \lambda\ (sel, int) . \emptyset))$
in $(\sigma[\ struct'\ /\ loc\ ], fl', label[\ structlbl'\ /\ loc\ ], loc)$
end*

$addcontext_A : Store_A \times Freelist \times Label_A \times Occ_A \times Point \times Occ_A \rightarrow Store_A \times Freelist \times Label_A \times Loc$
$addcontext_A\ (\sigma, fl, label, ptocc, retpt, retocc) =$
let $(fl', loc) = alloc\ (fl)$ **in**
let $struct' = (\mathbf{ordinary}, \mathbf{context}, \perp, (retpt, retocc), \perp)$ **and** $structlbl' = (\{ ptocc \}, \lambda\ (sel, idx) . \emptyset)$
in $(\sigma[\ struct'\ /\ loc\ ], fl', label[\ structlbl'\ /\ loc\ ], loc)$
end*

$updref : Store_A \times Label_A \times Occ_A \times Loc \times Sel \times Loc \rightarrow pwr(Store_A \times Label_A)$
$updref(\sigma, label, occ, src, sel, tgt) =$

if $(\exists\, type \in gettype_A(\sigma, src) : sel \in structDecls(type)) \vee maybe\_oftype(\sigma, src, \mathbf{env})$ **then**
  if $is\_summary(\sigma, src)$ **then** $\{ updsummary(\sigma, label, occ, src, sel, tgt) \}$
  else $\{ updordinary(\sigma, label, occ, src, sel, tgt) \}$
  fi
else $\emptyset$
fi

$updsummary : Store_A \times Label_A \times Occ_A \times Loc \times Sel \times Loc \rightarrow Store_A \times Label_A$
$updsummary(\sigma, label, occ, src, sel, tgt) =$

let $(kind, types, val, cxt, map) = \sigma(src)$ **and** $(structdefs, refdefs) = label(src)$
in
  let $idx =$ **if** $\exists\, j : map(sel, j) = tgt$ **then** j **else** any j such that $map(sel, j) = \perp$ **fi**
  in
    let $map' = map[tgt / (sel, idx)]$ **and** $refdefs' = refdefs[refdefs(sel, idx) \cup \{ occ \} / (sel, idx)]$
    in
      let $struct' = (kind, types, val, cxt, map')$ **and** $structlbl' = (structdefs, refdefs')$
      in $\{ (\sigma[struct' / src], label[structlbl' / src]) \}$
end*

$updordinary : Store_A \times Label_A \times Occ_A \times Loc \times Sel \times Loc \rightarrow Store_A \times Label_A$
$updordinary(\sigma, label, occ, src, sel, tgt) =$

let $(kind, types, val, cxt, map) = \sigma(src)$ **and** $(structdefs, refdefs) = label(src)$
in
  let $map' = \lambda\, (sel', idx') .\ sel' \neq sel \rightarrow map(sel', idx')\ [] \perp$
  and $refdefs' = \lambda\, (sel', idx') .\ sel' \neq sel \rightarrow refdefs(sel', idx')\ [] \emptyset$
  in
    let $struct' = (kind, types, val, cxt, map')$ **and** $structlbl' = (structdefs, refdefs')$
    in
      $newref((\sigma[struct' / src], label[structlbl' / src]), occ, src, sel, tgt)$
end*

$newref : Store_A \times Label_A \times Occ_A \times Loc \times Sel \times Loc \rightarrow Store_A \times Label_A$
$newref(\sigma, label, currocc, src, sel, tgt) =$

let $(kind, type, val, cxt, map) = \sigma(src)$ **and** $(structdefs, refdefs) = label(src)$
in
  let $map' = map\ [tgt / (sel, 0)]$ **and** $refdefs' = refdefs[\{ currocc \} / (sel, 0)]$
  in
    let $struct' = (kind, types, val, cxt, map')$ **and** $structlbl' = (structdefs, refdefs')$
    in
      $(\sigma[struct' / src], label[structlbl' / src])$
end*

$idexpr_A : Point \times Store_A \times Idexp \to pwr(Loc)$
$idexpr_A\,(pt,\ \sigma,\ id.sexp) =$
 let *globalEnv* be σ's global environment
 in $selexp_A\,(\sigma, globalEnv, (\ id \in localIds(pt, body') \to \_curr.id.sexp\ []\ id.sexp\ ))$
 end

$selexp_A : Store_A \times Loc \times (Ident + Sel)^* \to pwr(Loc)$
$selexp_A\,(\sigma,\ loc,\ sexp) =$
 $sexp \stackrel{?}{=} \varepsilon \to \{\ loc\ \}$
 [] **let** $(sel,\ sexp') = sexp$ **and** $map = getmap_A\,(\sigma,\ loc)$ **in**
  union_from i such that $map\,(sel,\ i) \neq \perp$: $selexp_A\,(\sigma,\ map\,(sel,\ i),\ sexp')$
  end

$is\_summary : Store_A \times Loc \to Bool = \lambda\,(\sigma, loc)\ .\ getkind_A\,(state, loc') \stackrel{?}{=} \mathbf{summary}$

$maybe\_atom : Store_A \times Loc \to Bool = \lambda\,(\sigma, loc)\ .\ maybe\_oftype\,(\sigma, loc, \mathbf{atom})$

$maybe\_not\_oftype : Store_A \times Loc \times Type \to Bool = \lambda\,(\sigma, loc, type)\ .\ \exists\ type' \in gettype_A\,(\sigma, loc) :\ type' \neq type$

$maybe\_oftype : \quad Store_A \times Loc \times Type \to Bool = \lambda\,(\sigma, loc, type)\ .\ type \in gettype_A\,(\sigma, loc)$

$getkind_A : Store_A \times Loc \to Kind = \lambda\,(\sigma, loc)\ .$ **let** $(knd, \_, \_, \_, \_) = \sigma(loc)$ **in** *knd* **end**
$gettype_A : Store_A \times Loc \to Type = \lambda\,(\sigma, loc)\ .$ **let** $(\_, typ, \_, \_, \_) = \sigma(loc)$ **in** *typ* **end**
$getval_A : Store_A \times Loc \to Atom = \lambda\,(\sigma, loc)\ .$ **let** $(\_, \_, val, \_, \_) = \sigma(loc)$ **in** *val* **end**
$getcxt_A : Store_A \times Loc \to Context_A = \lambda\,(\sigma, loc)\ .$ **let** $(\_, \_, \_, cxt, \_) = \sigma(loc)$ **in** *cxt* **end**
$getmap_A : Store_A \times Loc \to Selmap = \lambda\,(\sigma, loc)\ .$ **let** $(\_, \_, \_, \_, map) = \sigma(loc)$ **in** *map* **end**
$alloc : Freelist \to Loc \times Freelist = \lambda\,fl\ .$ **let** $(loc, fl') = fl$ **in** (loc, fl') **end**

Appendix 4. Abstraction and Subsumption Relations

This appendix formally the abstraction and subsumption relations introduced in Chapter 5. The first relation, $\rhd$, relates objects in the instrumented semantics to objects in the approximation semantics.

DEFINITION (***type abstraction***). Let a be a type, and *bset* a set of types. Then *bset abstracts* a, written $a \rhd bset$, iff $a \in bset$. □

DEFINITION (***value abstraction***). Let a and b be atoms. Then b *abstracts* a, written $a \rhd b$, iff $a = b$ or $b = \top^{AT}$. □

DEFINITION. Let $apxocc \in ApproxOcc$ be an approximate occurrence string. Then $L(apxocc)$ denotes the family of program-point strings denoted by the regular expression $apxocc$. □

DEFINITION. Let $apxoccset \in pwr(ApproxOcc)$ be a set of approximate occurrence string. Then $L(apxoccset)$ denotes the union, over all $apxocc$ in $apxoccset$, of $L(apxocc)$. □

DEFINITION (***occurrence string abstraction***). Let $occ \in Occ$, and $occ_A \in ApproxOcc$. Then occ_A *abstracts* occ, written $occ \rhd occ_A$, iff $occ \in L(occ_A)$. □

DEFINITION (***occurrence string abstraction***). Let $occ \in Occ$, and $occ_{*A} \in pwr(ApproxOcc)$. Then occ_{*A} *abstracts* occ, written $occ \rhd occ_{*A}$, iff $occ \in L(occ_{*A})$. □

DEFINITION (***context abstraction***). Let $cxt = (pt, occ) \in Context_I$, and $cxt_A = (pt_A, occ_A) \in Context_A$. Then cxt_A *abstracts* cxt, written $cxt \rhd cxt_A$, iff $pt = pt_A$ and $occ \rhd occ_A$. □

DEFINITION (***context abstraction***). Let $cxt = (pt, occ) \in Context_I$, and $cxt_{*A} \in pwr(Context_A)$. Then cxt_{*A} *abstracts* cxt, written $cxt \rhd cxt_{*A}$, iff there exists a $cxt_A \in cxt_{*A}$ such that $cxt \rhd cxt_A$. □

DEFINITION (***admissible map***). Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ be a map. Map f is ***admissible*** iff for all $loc \in Domain(f)$, and all sel_1 and $sel_2 \in Domain(f(loc))$,

* $f(loc, sel_1)$ is of the form $(\_, sel_1, \_)$.
* $f(loc, sel_1)$ is of the form $(loc', \_, \_) \Rightarrow f(loc, sel_2)$ is of the form $(loc', \_, \_)$. □

Intuitively, an admissible map is a map from structures and references to structures and references that preserves the type and the source node of every reference. Every f of the form $Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ defined below is assumed to be admissible.

DEFINITION (f_{struct}, f_{ref}). Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ be a map, and $loc \in Domain(f)$. Let $sel \in Sel$ be any selector in $Domain(f(loc))$ and $f(loc, sel) = (loc', sel', i')$. Then $f_{struct}(loc)$ denotes loc', and $f_{ref}(loc, sel)$ denotes (sel', i'). □

DEFINITION (***type, atom, ctxt, ref***). Let $\sigma \in Store_I$. Let $loc \in Domain(\sigma)$ be a location in σ. Let $(typ, atm, cxt, map) = \sigma(loc)$. Then ***type*** (loc) denotes typ; ***atom*** (loc) denotes atm; ***ctxt*** (loc) denotes cxt; and ***ref*** (loc, sel) denotes $map(sel)$. □

DEFINITION (***kind, type, atom, ctxt, ref***). Let $\sigma \in Store_A$ and $loc \in Domain(\sigma)$. (*N.B.*: the definitions of $Store_A$, $Label_A$, and the other domains of abstract objects are given in Appendix 3.) Let $(knd, typ, atm, cxt, map) = \sigma(loc)$. Then ***kind*** (loc) denotes knd; ***type*** (loc) denotes typ; ***atom*** (loc) denotes atm; ***ctxt*** (loc) denotes $ctxt$; and ***ref*** (loc, sel, i) denotes $map(sel, i)$. □

DEFINITION (***kind-preserving***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ map σ to σ_A. Then f is *kind-preserving* iff, for all *loc* and $loc' \in Domain(f)$,

* $kind(f_{struct}(loc)) = \textbf{ordinary} \Rightarrow f_{struct}(loc) \neq f_{struct}(loc')$. □

DEFINITION (***type-preserving***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ map σ to σ_A. Then f is *type-preserving* iff, for all $loc \in Domain(f)$,

* $type(loc) \rhd type(f_{struct}(loc))$. □

DEFINITION (***atom-preserving***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ map σ to σ_A. Then f is *atom-preserving* iff, for all $loc \in Domain(f)$,

* $type(loc) = \textbf{atom} \Rightarrow \textbf{atom} \in type(f_{struct}(loc)) \wedge atom(loc) \rhd atom(f_{struct}(loc))$. □

DEFINITION (***context-preserving***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ map σ to σ_A. Then f is *context-preserving* iff, for all $loc \in Domain(f)$,

* $type(loc) = \textbf{context} \Rightarrow \textbf{context} \in type(f_{struct}(loc)) \wedge ctxt(loc) \rhd ctxt(f_{struct}(loc))$. □

DEFINITION (***reference-preserving***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ map σ to σ_A. Then f is *reference-preserving* iff, for all $loc \in Domain(f)$ and all $sel \in Domain(f(loc))$,

* $f_{struct}(ref(loc, sel)) = ref(f_{struct}(loc), sel', i')$, where $(sel', i') = f_{ref}(loc, sel)$. □

DEFINITION (***store abstraction***). Let $\sigma \in Store_I$ and $\sigma_A \in Store_A$. Store σ_A *abstracts* σ, written $\sigma \rhd \sigma_A$, iff there exists a map $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ such that

* f maps every accessible structure and reference in σ into σ_A;
* f maps σ's global environment to σ_A's global environment; and
* f is kind-, type-, atom-, context-, and reference-preserving. □

DEFINITION (***labeled store abstraction***). Let $ls = (\sigma, label) \in Store_I \times Label$ and $ls_A = (\sigma_A, label_A) \in Store_A \times Label_A$. Labeled store ls_A *abstracts* ls, written $ls \rhd ls_A$, iff there exists a map $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ such that

* $\sigma \rhd \sigma_A$ by f;
* for all locations $loc \in Domain(f)$, $type(loc) \neq \textbf{atom} \Rightarrow label(loc) \rhd label_A(f_{struct}(loc))$; and
* for all locations $loc \in Domain(f)$ and all $sel \in Domain(f(loc))$, $label(loc, sel) \rhd label_A(f_{struct}(loc), sel', i')$, where $(sel', i') = f_{ref}(loc, sel)$. □

DEFINITION (***state abstraction***). Let $state = (pt, \sigma, fl, label, occ) \in State_I$ and $state_A = (pt_A, \sigma_A, fl_A, label_A, occ_A) \in State_A$. State $state_A$ *abstracts* $state$, written $state \rhd state_A$, iff $pt = pt_A$, $(\sigma, label) \rhd (\sigma_A, label_A)$, and $occ \rhd occ_A$. □

DEFINITION (***stateset abstraction***). Let $state_* \in pwr(State_I)$ and $state_{*A} \in pwr(State_A)$. Stateset $state_{*A}$ *abstracts* $state_*$, written $state_* \rhd state_{*A}$, iff for all $state \in state_*$ there exists a $state_A \in state_{*A}$ such that $state_I \rhd state_A$. □

The second relation, $\sqsubseteq$, relates objects in the approximation semantics to objects in the approximation semantics.

DEFINITION (***type subsumption***). Let *aset* and *bset* be sets of types. Then *bset subsumes aset*, written $aset \sqsubseteq bset$, iff $aset \subseteq bset$. □

DEFINITION (***value subsumption***). Let a and b be atoms. Then b *subsumes* a, written $a \sqsubseteq b$, iff either $a = b$ or $b = \top^{AT}$. □

DEFINITION (***occurrence string subsumption***). Let $apxocc$ and $apxocc' \in ApproxOcc$. Then $apxocc'$ *subsumes* $apxocc$, written $apxocc \sqsubseteq apxocc'$, iff $L(apxocc) \subseteq L(apxocc)$. □

DEFINITION (***occurrence string subsumption***). Let $apxocc_*$ and $apxocc'_* \in pwr(ApproxOcc)$. Then $apxocc'_*$ *subsumes* $apxocc_*$, written $apxocc_* \sqsubseteq apxocc'_*$, iff $L(apxocc_*) \subseteq L(apxocc'_*)$. □

DEFINITION (***context subsumption***). Let $cxt = (pt, occ)$ and $cxt' = (pt', occ') \in Context_A$. Then cxt' *subsumes* cxt, written $cxt \sqsubseteq cxt'$, iff $pt = pt'$ and $occ \sqsubseteq occ'$. □

DEFINITION (***context subsumption***). Let cxt_* and $cxt'_* \in pwr(Context_A)$. Then cxt'_* *subsumes* cxt_*, written $cxt_* \sqsubseteq cxt'_*$, iff for all $cxt \in cxt_*$ there exists a $cxt' \in cxt'_*$ such that $cxt' \sqsubseteq cxt'$. □

DEFINITION (***admissible map***). Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ be a map. Map f is ***admissible*** iff for all $loc \in Domain(f)$, all sel_1 and $sel_2 \in Domain(f(loc))$, all $i_1 \in Domain(f(loc, sel_1))$, and all $i_2 \in Domain(f(loc, sel_2))$,

* $f(loc, sel_1, i_1)$ is of the form $(\_, sel_1, \_)$.
* $f(loc, sel_1, i_1)$ is of the form $(loc', \_, \_) \Rightarrow f(loc, sel_2, i_2)$ is of the form $(loc', \_, \_)$. □

Every f of the form $Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ defined below is assumed to be admissible.

DEFINITION (f_{struct}, f_{ref}). Let $f: Loc \rightarrow Sel \rightarrow (Loc \times Sel \times Int)$ be a map, and $loc \in Domain(f)$. Let $sel \in Sel$ be any selector in $Domain(f(loc))$, i be any integer in $Domain(f(loc, sel))$, and $f(loc, sel) = (loc', sel', i')$. Then $f_{struct}(loc)$ denotes loc', and $f_{ref}(loc, sel, i)$ denotes (sel', i'). □

DEFINITION (***kind-preserving***). Let σ and $\sigma' \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ map σ to σ'. Then f is *kind-preserving* iff, for all loc and $loc' \in Domain(f)$,

* $kind(f_{struct}(loc)) = \textbf{ordinary} \Rightarrow kind(loc) = \textbf{ordinary} \wedge f_{struct}(loc) \neq f_{struct}(loc')$. □

DEFINITION (***type-preserving***). Let σ and $\sigma' \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ map σ to σ'. Then f is *type-preserving* iff, for all $loc \in Domain(f)$,

* $type(loc) \sqsubseteq type(f_{struct}(loc))$. □

DEFINITION (***atom-preserving***). Let σ and $\sigma' \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ map σ to σ'. Then f is *atom-preserving* iff, for all $loc \in Domain(f)$,

* $\textbf{atom} \in type(loc) \Rightarrow \textbf{atom} \in type(f_{struct}(loc)) \wedge atom(loc) \sqsubseteq atom(f_{struct}(loc))$. □

DEFINITION (***context-preserving***). Let σ and $\sigma' \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ map σ to σ'. Then f is *context-preserving* iff, for all $loc \in Domain(f)$,

* $\textbf{context} \in type(loc) \Rightarrow \textbf{context} \in type(f_{struct}(loc)) \wedge ctxt(loc) \sqsubseteq ctxt(f_{struct}(loc))$. □

DEFINITION (***reference-preserving***). Let σ and $\sigma' \in Store_A$. Let $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ map σ to σ'. Then f is *reference-preserving* iff, for all $loc \in Domain(f)$, all $sel \in Domain(f(loc))$, and all $i \in Domain(f(loc, sel))$,

* $f_{struct}(ref(loc, sel, i)) = ref(f_{struct}(loc), sel', i')$, where $(sel', i') = f_{ref}(loc, sel, i)$. □

DEFINITION (***store subsumption***). Let σ and $\sigma' \in Store_A$. Store σ' *subsumes* σ, written $\sigma \sqsubseteq \sigma'$, iff there exists a map $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ such that

* f maps every accessible structure and reference in σ into σ';
* f maps σ's global environment to σ''s global environment; and
* f is kind-, type-, atom-, context-, and reference-preserving. □

DEFINITION (***labeled store subsumption***). Let $ls = (\sigma, label)$ and $ls' = (\sigma', label') \in Store_A \times Label_A$ be labeled stores. Labeled store ls' ***subsumes*** ls, written $ls \sqsubseteq ls'$, iff there exists a map $f: Loc \rightarrow Sel \rightarrow Int \rightarrow (Loc \times Sel \times Int)$ such that

* $\sigma \sqsubseteq \sigma'$ by f;
* for all locations $loc \in Domain(f)$, $type(loc) \neq \{ atom \} \Rightarrow label(loc) \sqsubseteq label'(f_{struct}(loc))$; and
* for all locations $loc \in Domain(f)$, all $sel \in Domain(f(loc))$, and all $i \in Domain(f(loc, sel))$, $label(loc, sel, i) \sqsubseteq label_A(f_{struct}(loc), sel', i')$, where $(sel', i') = f_{ref}(loc, sel, i)$. □

DEFINITION (***state subsumption***). Let $state = (pt, \sigma, fl, label, occ)$ and $state' = (pt', \sigma', fl', label', occ') \in State_A$. State $state'$ *subsumes* $state$, written $state \sqsubseteq state'$, iff $pt = pt'$, $(\sigma, label) \sqsubseteq (\sigma', label')$, and $occ \sqsubseteq occ'$. □

DEFINITION (***stateset subsumption***). Let $state_*$ and $state'_* \in pwr(State_A)$. Stateset $state'_*$ *subsumes* $state_*$, written $state_* \sqsubseteq state'_*$, iff for all $state \in state_*$ there exists a $state' \in state'_*$ such that $state \sqsubseteq state'$. □

Appendix 5. The Monotonicity of $evalPt_A$

This appendix demonstrates that $\mathbf{MA}_{\mathcal{H}}$'s state transition function is monotonic w.r.t. $\sqsubseteq$. The proof is by a series of lemmas that characterize the various components of $evalPt_A$'s definition.

LEMMA A.1 (***monotonicity of getkind$_A$, etc.***). If $\sigma \sqsubseteq \sigma'$ by f and $loc \in Domain(f)$, then

* $getkind(\sigma, loc) \sqsubseteq getkind(\sigma', f_{struct}(loc))$;
* $gettype(\sigma, loc) \sqsubseteq gettype(\sigma', f_{struct}(loc))$;
* $\mathbf{atom} \in gettype(\sigma, loc) \Rightarrow getval(\sigma, loc) \sqsubseteq getval(\sigma', f_{struct}(loc))$;
* $\mathbf{context} \in gettype(\sigma, loc) \Rightarrow getcxt(\sigma, loc) \sqsubseteq getcxt(\sigma', f_{struct}(loc))$;
* For all $(sel, i) \in Domain(f(loc))$, $f_{struct}(map(sel, i)) = map'(sel', i')$, where $map = getmap(\sigma, loc)$, $map' = getmap(\sigma', f_{struct}(loc))$, and $(sel', i') = f_{ref}(loc, sel, i)$.

PROOF. Immediate from the definition of $\sqsubseteq$. (*N.B.*: f_{struct} and f_{ref}, which identify the structure-specific and reference-specific parts of an embedding, are defined in Appendix 4.) □

LEMMA A.2 (***monotonicity of maybe_atom, etc.***). If $\sigma \sqsubseteq \sigma'$ by f and $loc \in Domain(f)$, then

* $maybe\_atom(\sigma, loc) \Rightarrow maybe\_atom(\sigma', f_{struct}(loc))$;
* $maybe\_oftype(\sigma, loc, type) \Rightarrow maybe\_oftype(\sigma', f_{struct}(loc), type)$;
* $maybe\_not\_oftype(\sigma, loc, type) \Rightarrow maybe\_not\_oftype(\sigma', f_{struct}(loc), type)$;
* $is\_summary(\sigma, loc) \Rightarrow is\_summary(\sigma', f_{struct}(loc))$.

PROOF. Immediate from Lemma A.1 and the definitions of these four functions. □

LEMMA A.3. (***monotonicity of selexp$_A$***). If $\sigma \sqsubseteq \sigma'$ by f and $loc \in Domain(f)$, then, for all $sexp$, $loc_{tgt} \in selexp_A(\sigma, loc, sexp) \Rightarrow f_{struct}(loc_{tgt}) \in selexp_A(\sigma, f_{struct}(loc), sexp)$.

PROOF. By induction on the length of $sexp$.

If $sexp = \varepsilon$, the empty selector string, then $selexp_A(\sigma, loc, sexp) = loc$ and $selexp_A(\sigma, f_{struct}(loc), sexp) = f_{struct}(loc)$.

If $sexp$ is nonempty, let $sexp$ be of the form $sel.selexpr$, where $selexpr$ is possibly empty, and assume that Lemma A.3 holds for selector expressions of length $selexpr$. If loc lacks a selector sel, then the lemma is immediate. Otherwise, assume that the integers $i_1 \cdots i_k$ denote valid selectors of type sel at loc. Let these selectors denote the locations $loc_1 \cdots loc_k$, respectively. Then $selexp_A(\sigma, loc, sel.selexpr)$ is defined to be $selexp_A(\sigma, loc_1, selexpr) \cup \cdots \cup selexp_A(\sigma, loc_k, selexpr)$. However, by the definition of $\sqsubseteq$, $f_{struct}(loc)$ has references of type sel to $f_{struct}(loc_1) \cdots f_{struct}(loc_k)$. Then $selexp_A(\sigma, loc, sel.selexpr)$ is a subset of $selexp_A(\sigma', f_{struct}(loc_1), selexpr) \cup \cdots \cup selexp_A(\sigma', f_{struct}(loc_k), selexpr)$. Lemma A.3 now follows from the induction hypothesis. □

LEMMA A.4 (***monotonicity of idexpr$_A$***). If $\sigma \sqsubseteq \sigma'$ by f and $loc \in Domain(f)$, then, for all $sexp$, $loc_{tgt} \in idexpr_A(pt, \sigma, sexp) \Rightarrow f_{struct}(loc_{tgt}) \in idexpr_A(pt, \sigma', sexp)$.

PROOF. Immediate from Lemma A.3 and the fact that the subsumption relation maps the global environment of σ to the global environment of σ'. □

LEMMA A.5 (***monotonicity of newref***). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$. Let src and tgt be in $Domain(f)$. Assume that src has no selectors of type sel. Then

$newref(\sigma, label, occ, src, sel, tgt) \sqsubseteq newref(\sigma, label, occ', f_{struct}(src), sel, f_{struct}(tgt))$.

PROOF. Immediate from the definition of $\sqsubseteq$. The new f maps $(src, sel, 0)$ to $(f_{struct}(src), sel, 0)$ and is otherwise unchanged. □

LEMMA A.6 (***monotonicity of updordinary***). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$. Let src and tgt be in $Domain(f)$. Assume that $f_{struct}(src)$ is an ordinary node. Then $updordinary(\sigma, label, occ, src, sel, tgt) \sqsubseteq updordinary(\sigma', label', occ', f_{struct}(src), sel, f_{struct}(tgt))$.

PROOF. Let $struct = (kind, types, val, cxt, map) = \sigma(src)$ and $(structdefs, refdefs) = label(src)$. Let $struct_{new}$ be the updated $struct$ obtained by setting all sel components of map to $\perp$. Let $label_{new}$ be the updated $label$ obtained by setting all sel components of $label$ to the empty set. Let $struct' = (kind', types', val', cxt', map') = \sigma(f_{struct}(src))$, and $(structdefs', refdefs') = label(f_{struct}(src))$. Let $struct'_{new}$ be the updated $struct'$ obtained by setting all sel components of map' to $\perp$. Let $label'_{new}$ be the updated $label'$ obtained by setting all sel components of $label'$ to the empty set. Since $f_{struct}(src)$ is an ordinary node, src, by definition of subsumption, is the only node that is mapped to $f_{struct}(src)$. Hence, by the definition of subsumption, $(\sigma_{new}, label_{new}) \sqsubseteq (\sigma'_{new}, label'_{new})$. Lemma A.6 now follows from Lemma A.5, since *updordinary* calls *newref* to add the new reference. □

LEMMA A.7 (***monotonicity of updsummary***). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$. Let src and tgt be in $Domain(f)$. Then $updsummary(\sigma, label, occ, src, sel, tgt) \sqsubseteq updsummary(\sigma', label', occ', f_{struct}(src), sel, f_{struct}(tgt))$.

PROOF. The new f maps the edge from sel to tgt added to σ to the edge from $f_{struct}(src)$ to $f_{struct}(tgt)$ added to σ'. The updated $label'$ subsumes the updated $label$ because the new edge from $f_{struct}(src)$ to $f_{struct}(tgt)$ is labeled $\{ occ' \}$, and the new edge from src to tgt is labeled $\{ occ \}$. □

LEMMA A.8 (***monotonicity of updref***). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$. Let src and tgt be in $Domain(f)$. Then $updref(\sigma, label, occ, src, sel, tgt) \sqsubseteq updref(\sigma', label', occ', f_{struct}(src), sel, f_{struct}(tgt))$.

PROOF. If src can support a selector of type sel, then $f_{struct}(src)$, by the definition of $\sqsubseteq$, must also support selectors of type sel. The proof now proceeds by an analysis of the kind of src.

If src is a summary structure, then $f_{struct}(src)$ must also be a summary structure. Lemma A.8 now follows from the monotonicity of *updsummary* (Lemma A.7).

If src and $f_{struct}(src)$ are ordinary structures, Lemma A.8 follows from the isontonicity of *updordinary* (Lemma A.6).

If src is an ordinary structure and $f_{struct}(src)$ a summary structure, then $updref(\sigma, \cdots)$ uses *updordinary* to add an edge r to σ of type sel labeled $currocc$. Also, $updref(\sigma', \cdots)$ uses *updsummary* to add an edge r' to σ' of type sel labeled $currocc'$. The embedding from the updated $(\sigma, label)$ to the updated $(\sigma', label')$ is an extension of f that maps r to r'. □

LEMMA A.9 (***monotonicity of addcontext$_A$***). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$, $retpt = retpt'$, and $retocc \sqsubseteq retocc'$. Let $(\sigma_{new}, fl_{new}, label_{new}, loc_{new}) = addcontext_A(\sigma, fl, label, occ, retpt, retocc)$. Let $(\sigma'_{new}, fl'_{new}, label'_{new}, loc'_{new}) = addcontext_A(\sigma', fl', label', occ', retpt', retocc')$. Then $(\sigma_{new}, label_{new}) \sqsubseteq (\sigma'_{new}, label'_{new})$ by an f that sends loc_{new} to loc'_{new}.

PROOF. Immediate from the definition of $addcontext_A$. □

LEMMA A.10 (***monotonicity of addstruct***$_A$). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$ and $type \sqsubseteq type'$. Let $(\sigma_{new}, fl_{new}, label_{new}, loc_{new}) = addstruct_A(\sigma, fl, label, occ, type)$. Let $(\sigma'_{new}, fl'_{new}, label'_{new}, loc'_{new}) = addstruct_A(\sigma', fl', label', occ', type')$. Then $(\sigma_{new}, label_{new}) \sqsubseteq (\sigma'_{new}, label'_{new})$ by an f that sends loc_{new} to loc'_{new}.

PROOF. Immediate from the definition of $addstruct_A$. □

LEMMA A.11 (***monotonicity of addatom***$_A$). Let $(\sigma, label) \sqsubseteq (\sigma', label')$ by f. Let $occ \sqsubseteq occ'$ and $value \sqsubseteq value'$. Let $(\sigma_{new}, fl_{new}, label_{new}, loc_{new}) = addatom_A(\sigma, fl, label, occ, value)$. Let $(\sigma'_{new}, fl'_{new}, label'_{new}, loc'_{new}) = addatom_A(\sigma', fl', label', occ', value')$. Then $(\sigma_{new}, label_{new}) \sqsubseteq (\sigma'_{new}, label'_{new})$ by an f that sends loc_{new} to loc'_{new}.

PROOF. Immediate from the definition of $addatom_A$. □

LEMMA A.12 (***monotonicity of simplexp***$_A$). Let $state = (pt, \sigma, fl, label, occ)$ and $state' = (pt', \sigma', fl', label', occ')$ be states such that $state \sqsubseteq state'$ w.r.t. an f that maps σ onto σ'. Let $result_* = simplexp_A(state, exp)$ and $result'_* = simplexp_A(state', exp)$. Let $(\sigma_{new}, fl_{new}, label_{new}, loc_{new})$ be a member of $result_*$. Then there exists a $(\sigma'_{new}, fl'_{new}, label'_{new}, loc'_{new})$ in $result'_*$ such that $(\sigma_{new}, label_{new}) \sqsubseteq (\sigma'_{new}, label'_{new})$ by a (possibly extended) f that sends loc_{new} to loc'_{new}.

PROOF. The proof is by an analysis of the type of *exp*.

If *exp* is an identifier expression, then let $loc_* = idexpr_A(pt, \sigma, exp)$ and $loc'_* = idexpr_A(pt, \sigma', exp)$. By the monotonicity of $idexpr_A$ (Lemma A.4), $loc \in loc_* \Rightarrow f_{struct}(loc) \in loc'_*$. Lemma A.12 now follows from the fact that $selexp_A$ produces $result_*$ by pairing every $loc \in loc_*$ with $(\sigma, fl, label)$, and $result'_*$ by pairing every $loc' \in loc'_*$ with $(\sigma', fl', label')$.

If *exp* is an atom, then Lemma A.12 follows immediately from the monotonicity of $addatom_A$ (Lemma A.11). The extended f maps the new atom in σ to the new atom in σ'.

If *exp* is a context, then Lemma A.12 follows immediately from the monotonicity of $addcontext_A$ (Lemma A.9). The extended f maps the new context in σ to the new context in σ'.

If *exp* is a **new**(*type*) instruction, then let $(\sigma_1, fl_1, label_1, loc_1) = addstruct_A(\sigma, fl, label, ptocc, type)$, and $(\sigma'_1, fl'_1, label'_1, loc'_1) = addstruct_A(\sigma', fl', label', ptocc', type')$. By the monotonicity of $addstruct_A$ (Lemma A.10), $(\sigma_1, label_1) \sqsubseteq (\sigma_2, label_2)$ by an extended f that sends loc_1 to loc'_1. If $type$ = **env**, then Lemma A.12 is immediate. Otherwise, the proof of the **new** case is completed by using the monotonicity of $addatom_A$ (Lemma A.11) and *newref* (Lemma A.5) to argue that adding a set of new references at loc_1 and loc'_1 to new **nil** atoms preserves the embedding from the updated $(\sigma_1, label_1)$ to the updated $(\sigma'_1, label'_1)$.

Finally, if *exp* is the primitive operator $op(exp_1, \cdots, exp_n)$, then the definition of $\mathcal{H}$ ensures that each of the exp_i's are either atoms or identifier expressions that denote atoms. An induction on n can be used to show that if $simplexp_A(state, exp)$ invokes op on a set of atoms $loc_1 \cdots loc_n$ then $simplexp_A(state', exp)$ invokes op on a set of atoms $loc'_1 \cdots loc'_n$, where each of the loc'_i has a value that subsumes the value of the corresponding loc_1. Lemma A.12 then follows from the assumption that primitive operands are monotonic in all arguments. □

LEMMA A.13 (***monotonicity of $cond_T$, $cond_F$***). Let $\sigma \sqsubseteq \sigma'$ by f. Then $cond_T(pt, \sigma, cexp)$ implies $cond_T(pt, \sigma', cexp)$, and $cond_F(pt, \sigma, cexp)$ implies $cond_F(pt, \sigma', cexp)$.

PROOF. This lemma is proved on the depth to which *not* operators are nested in *cexp*.

BASIS (*cexp* contains no negations). The proof is by an analysis of the type of *cexp*.

If *cexp* is the predicate $typeOf(exp, type)$, then let $locs = idexpr_A(pt, \sigma, exp)$ and $locs' = idexpr_A(pt, \sigma', exp)$. If there exists a $loc \in locs$ that satisfies *maybe_oftype*, then, by the monotonicity of $idexpr_A$ (Lemma A.4), $f_{struct}(loc)$ is in $locs'$. By the monotonicity of *maybe_oftype* (Lemma A.2), $f_{struct}(loc)$ also satisfies *maybe_oftype*. Similarly, if there exists a $loc \in locs$ that satisfies *maybe_not_oftype*, then $f_{struct}(loc)$ is in $locs'$. By the monotonicity of *maybe_not_oftype* (Lemma A.2). $f_{struct}(loc)$ also satisfies *maybe_not_oftype*.

If *cexp* is the predicate $Eq(exp_1, exp_2)$, then let $locs_1 = idexpr_A(pt, \sigma, exp_1)$, $locs_2 = idexpr_A(pt, \sigma, exp_2)$, $locs'_1 = idexpr_A(pt, \sigma', exp_1)$, and $locs'_2 = idexpr_A(pt, \sigma', exp_2)$. If σ satisfies *Eq*, then $locs_1 \cap locs_2$ contains some location *loc*. Then, by the monotonicity of $idexpr_A$, $locs'_1 \cap locs'_2$ must contain $f_{struct}(loc)$. Hence, σ' satisfies *Eq*. If σ satisfies the converse of *Eq*, the intersection of $locs_1$ or $locs_2$ either contains a summary structure loc_0, or there exist a $loc_1 \in locs_1$ and a $loc_2 \in locs_2$ such that $loc_1 \neq loc_2$. In the first case, the monotonicity of *is_summary* ensures that $f_{struct}(loc_0)$ is also a summary structure (Lemma A.2). In the second case, either loc_1 and loc_2 are mapped to different structures, satisfying the converse of *Eq*, or loc_1 and loc_2 are mapped to the same summary structure, which also satisfies the converse of *Eq*.

If *cexp* is a comparison operator, then Lemma A.13 follows from the monotonicity of the map from atoms to atoms.

INDUCTION HYPOTHESIS. Lemma A.14 holds when *cexp* contains k negations.

INDUCTION STEP. Let *cexp* be of the form *not*(*cexp'*). Then the assertion that $cond_T(pt, \sigma, cexp)$ implies $cond_T(pt, \sigma', cexp)$, by the definition of $cond_T$, is equivalent to the assertion that $cond_F(pt, \sigma, cexp')$ implies $cond_F(pt, \sigma', cexp')$. Lemma A.13 now follows from the induction hypothesis. A similar argument shows that Lemma A.13 holds for $cond_F$ when *cexp* is nested to depth $k+1$. □

LEMMA A.14 (***monotonicity of $evalPt_A$***). Let $state = (pt, \sigma, fl, label, occ)$ and $state' = (pt', \sigma', fl', label', occ')$ be states such that $state \sqsubseteq state'$ w.r.t. an f that maps σ onto σ'. Then $evalPt_A(state, exp) \sqsubseteq evalPt_A(state', exp)$.

PROOF. The proof is by an analysis of the type of *pt*.

If *pt* is the predicate *If* (*cexp*), then let $bool_T = cond_T(pt, \sigma, cexp)$, $bool_F = cond_F(pt, \sigma, cexp)$, $bool'_T = cond_T(pt, \sigma', cexp)$, and $bool'_F = cond_F(pt, \sigma', cexp)$. By the monotonicity of $cond_T$ and $cond_F$ (Lemma A.13), $bool_T$ implies $bool'_T$ and $bool_F$ implies $bool'_F$. Lemma A.14 now follows immediately from the definition of $evalPt_A$.

The proof of Lemma A.14 for when *pt* is the predicate *While* (*cexp*) is similar.

If *pt* is the statement *Assign*(*lexp.sel*, *rexp*), then let $rresult_* = simplexp_A(pt, state, rexp)$ and $rresult'_* = simplexp_A(pt, state', rexp)$. Let $(\sigma_r, fl_r, label_r, rvloc)$ be an arbitrary element of $rresult_*$.

By the monotonicity of $simplexp_A$ (Lemma A.12), $rresult'_*$ contains a $(\sigma'_r, fl'_r, label'_r, rvloc')$ such that $(\sigma_r, label_r) \sqsubseteq (\sigma'_r, label'_r)$ by an f' that maps $rvloc$ to $rvloc'$.

Let $lresult_* = idexpr_A(pt, \sigma_r, lexp)$ and $lresult'_* = idexpr_A(pt, \sigma'_r, lexp)$. Let $lvloc$ be an arbitrary element of $lresult_*$. By the monotonicity of $idexpr_A$ (Lemma A.4), $f'_{struct}(lvloc) \in lresult'_*$.

Let $assign_* = updref(\sigma_r, label_r, ptocc, lvloc, sel, rvloc)$ and $assign'_* = updref(\sigma'_r, label'_r, ptocc', f'_{struct}(lvloc), sel, rvloc')$. Let $(\sigma_a, label_a)$ be an arbitrary element of $assign_*$. By the monotonicity of *updref* (Lemma A.8), $assign'_*$ contains a $(\sigma'_a, label'_a)$ that subsumes $(\sigma_a, label_a)$.

Lemma A.14 now follows from the observation that $evalPt_A(state)$ returns $(nextpt, \sigma_a, fl_a, label_a, occ)$ only if $evalPt_A(state')$ returns $(nextpt, \sigma'_a, fl'_a, label'_a, occ)$.

If *pt* is a **call** statement, then Lemma A.14 follows immediately from the definition of $evalPt_A$.

If *pt* is a **return** statement, let $ctxloc_* = idexpr_A(pt, \sigma, \_curr.\_callctxt)$ and $ctxloc'_* = idexpr_A(pt, \sigma', \_curr.\_callctxt)$. Let *cloc* be an arbitrary element of $ctxloc_*$. By the monotonicity of $idexpr_A, f_{struct}(cloc) \in ctxloc'_*$.

Let $ctxloc_* = getcxt(\sigma, cloc)$ and $ctxloc'_* = getcxt(\sigma', f_{struct}(cloc))$. Let $(retpt, apxocc)$ be an arbitrary element of $ctxloc_*$. By the monotonicity of *getcxt* (Lemma A.1), $ctxloc'_*$ contains a $(retpt, apxocc')$ such that $apxocc \sqsubseteq apxocc'$.

Let $prevenv_* = idexpr_A(pt, \sigma, \_curr.\_callctxt)$ and $prevenv'_* = idexpr_A(pt, \sigma', \_curr.\_callctxt)$. Let *eloc* be an arbitrary element of $prevenv_*$. By the monotonicity of $idexpr_A, f_{struct}(eloc) \in prevenv'_*$.

By the definition of subsumption, *eloc*'s creation-point-label is a subset of $f_{struct}(eloc)$'s. If *eloc* is not consistent with *retpt*, then Lemma A.14 follows immediately. Assume, therefore, that *eloc* is consistent with *retpt*. Then the hypothesis that *state* and *state'* have comparable label functions implies that $f_{struct}(eloc)$ is consistent with *retpt*.

Let *gEnv* denote σ's global environment. By the definition of subsumption, the global environment of σ' is $f_{struct}(gEnv)$. Let $new_* = updref(\sigma, label, ptocc, gEnv, \_curr, eloc)$ and $new'_* = updref(\sigma', label', ptocc', f_{struct}(gEnv), \_curr, f_{struct}(eloc))$. Let $(\sigma_r, label_r)$ be an arbitrary element of new_*. By the monotonicity of *updref*, new'_* contains a $(\sigma'_r, label'_r)$ such that $(\sigma_r, label_r) \sqsubseteq (\sigma'_r, label'_r)$. Lemma A.14 now follows from the observation that $evalPt_A(state)$ returns $(retpt, \sigma_r, fl, label_r, apxocc)$ only if $evalPt_A(state')$ returns $(retpt, \sigma'_r, fl', label'_r, apxocc')$.

Finally, if *pt* is an initialization statement, then Lemma A.14 follows immediately from the definition of $evalPt_A$. □

Appendix 6. The Congruence of $evalPt_I$ and $evalPt_A$

This appendix demonstrates that $\mathbf{MA}_{\mathcal{H}}$'s state-transition function abstracts that of $\mathbf{MS}_{\mathcal{H}}$.

LEMMA A.1 (***congruence of gettype and gettype$_A$, etc.***). If $\sigma \rhd \sigma_A$ by a map f and $loc \in Domain(f)$, then

* $gettype(\sigma, loc) \rhd gettype_A(\sigma_A, f_{struct}(loc))$;
* **atom** $= gettype(\sigma, loc) \Rightarrow getval(\sigma, loc) \rhd getval_A(\sigma_A, f_{struct}(loc))$;
* **context** $= gettype(\sigma, loc) \Rightarrow getcxt(\sigma, loc) \rhd getcxt_A(\sigma_A, f_{struct}(loc))$;
* For all $sel \in Domain(f(loc))$, $f_{struct}(map(sel)) = map'(sel', i')$, where $map = getmap(\sigma, loc)$, $map' = getmap(\sigma_A, f_{struct}(loc))$, and $(loc', sel', i') = f_{ref}(loc, sel)$.

PROOF. Immediate from the definition of $\rhd$. (*N.B.*: f_{struct} and f_{ref}, which identify the structure-specific and reference-specific parts of an embedding, are defined in Appendix 4.) □

LEMMA A.2. (***congruence of selexp and selexp$_A$***). If $\sigma \rhd \sigma_A$ by a map f, and $loc \in Domain(f)$, then, for all $sexp$, $loc_{tgt} = selexp(\sigma, loc, sexp) \Rightarrow f_{struct}(loc_{tgt}) \in selexp_A(\sigma, f_{struct}(loc), sexp)$.

PROOF. By induction on the length of $sexp$.

If $sexp = \varepsilon$, the empty selector string, then $selexp(\sigma, loc, sexp) = loc$ and $selexp_A(\sigma, f_{struct}(loc), sexp) = f_{struct}(loc)$.

If $sexp$ is nonempty, let $sexp$ be of the form $sel.selexpr$, where $selexpr$ is possibly empty, and assume that Lemma A.2 holds for selector expressions of length $selexpr$. If loc lacks a selector sel, then the lemma is immediate. Otherwise, assume that the selector of type sel at loc references loc'. Then, by the definition of $\rhd$, $f_{struct}(loc)$ has a reference of type sel to $f_{struct}(loc')$. Lemma A.2 now follows from the induction hypothesis. □

COROLLARY 1. If $\sigma \rhd \sigma_A$ by a map f, and the evaluation of $selexp(\sigma, loc, sexp)$ accesses a structure or reference at loc', then the evaluation of $selexp_A(\sigma_A, f_{struct}(loc), sexp)$ accesses the structure or corresponding reference at $f_{struct}(loc')$. □

COROLLARY 2. If $(\sigma, label) \rhd (\sigma_A, label_A)$ by a map f, and the evaluation of $selexp(\sigma, loc, sexp)$ accesses a structure or reference labeled op, then the evaluation of $selexp_A(\sigma_A, f_{struct}(loc), sexp)$ accesses a structure or reference whose label subsumes op.

PROOF. Immediate from corollary 1 and the definition of $\rhd$. □

LEMMA A.3 (***congruence of idexpr and idexpr$_A$***). If $\sigma \rhd \sigma_A$ by a map f, and $loc \in Domain(f)$, then, for all $sexp$, $loc_{tgt} = idexpr(pt, \sigma, sexp) \Rightarrow f_{struct}(loc_{tgt}) \in idexpr_A(pt, \sigma_A, sexp)$.

PROOF. Immediate from Lemma A.2 and the fact that the abstraction relation maps the global environment of σ to the global environment of σ_A. □

COROLLARY 1. If $\sigma \rhd \sigma_A$ by a map f, and the evaluation of $idexpr(pt, \sigma, sexp)$ accesses a structure or reference at location loc, then the evaluation of $idexpr_A(pt, \sigma_A, sexp)$ accesses the structure or the corresponding reference at $f_{struct}(loc)$. □

COROLLARY 2. If $(\sigma, label) \rhd (\sigma_A, label_A)$ by a map f, and the evaluation of $idexpr(pt, \sigma, sexp)$ accesses a structure or reference labeled op, then the evaluation of $idexpr_A(pt, \sigma_A, sexp)$ accesses a structure or reference whose label abstracts op.

PROOF. Immediate from corollary 1 and the definition of $\rhd$. □

LEMMA A.4 (***congruence of addcontext and addcontext$_A$***). Let $(\sigma, label) \rhd (\sigma_A, label_A)$ by f. Let $occ \rhd occ_A$, $retpt = retpt_A$, and $retocc \rhd retocc_A$. Let $(\sigma', fl', label', loc') = addcontext\,(\sigma, fl, label, occ, retpt, retocc)$. Let $(\sigma'_A, fl'_A, label'_A, loc'_A) = addcontext_A\,(\sigma_A, fl_A, label_A, occ_A, retpt_A, retocc_A)$. Then $(\sigma', label') \rhd (\sigma'_A, label'_A)$ by an f that sends loc' to loc'_A.

PROOF. Immediate from the definitions of *addcontext* and *addcontext$_A$*. □

LEMMA A.5 (***congruence of addstruct and addstruct$_A$***). Let $(\sigma, label) \rhd (\sigma_A, label_A)$ by f. Let $occ \rhd occ_A$ and $type \rhd type_A$. Let $(\sigma', fl', label', loc') = addstruct\,(\sigma, fl, label, occ, type)$. Let $(\sigma'_A, fl'_A, label'_A, loc'_A) = addstruct_A\,(\sigma_A, fl_A, label_A, occ_A, type_A)$. Then $(\sigma', label') \rhd (\sigma'_A, label'_A)$ by an f that sends loc' to loc'_A.

PROOF. Immediate from the definitions of *addstruct* and *addstruct$_A$*. □

LEMMA A.6 (***congruence of addatom and addatom$_A$***). Let $(\sigma, label) \rhd (\sigma_A, label_A)$ by f. Let $occ \rhd occ_A$ and $value \rhd value_A$. Let $(\sigma', fl', label', loc') = addatom\,(\sigma, fl, label, occ, value)$. Let $(\sigma_A, fl_A, label_A, loc_A) = addatom_A\,(\sigma_A, fl_A, label_A, occ_A, value_A)$. Then $(\sigma_{new}, label_{new}) \rhd (\sigma_A, label_A)$ by an f that sends loc' to loc'_A.

PROOF. Immediate from the definitions of *addatom* and *addatom$_A$*. □

LEMMA A.7 (***congruence of simplexp and simplexp$_A$***). Let $state = (pt, \sigma, fl, label, occ)$ and $state_A = (pt, \sigma_A, fl_A, label_A, occ_A)$ be states such that $state \rhd state_A$ w.r.t. an f that maps σ onto σ_A. Let $(\sigma', fl', label', loc') = simplexp\,(state, exp)$ and $result_{*A} = simplexp_A\,(state_A, exp)$. Then there exists a $(\sigma'_A, fl'_A, label'_A, loc'_A)$ in $result_{*A}$ such that $(\sigma', label') \rhd (\sigma_A, label_A)$ by a (possibly extended) f that sends loc' to loc'_A.

PROOF. The proof is by an analysis of the type of *exp*.

If *exp* is an identifier expression and $idexpr\,(pt, \sigma, exp)$ is undefined, then the lemma is immediate. Otherwise, let $loc' = idexpr\,(pt, \sigma, exp)$ and $loc'_* = idexpr_A\,(pt, \sigma_A, exp)$. Then, by the congruence of *idexpr* and *idexpr$_A$* (Lemma A.3), $f_{struct}\,(loc') \in loc'_*$. Lemma A.7 now follows from the fact that *selexp* returns $(\sigma_A, fl_A, label_A, f_{struct}\,(loc'))$ in $result_{*A}$.

If *exp* is an atom, then Lemma A.7 follows immediately from the congruence of *addatom* and *addatom$_A$* (Lemma A.6). The extended f maps the new atom in σ to the new atom in σ_A.

If *exp* is a context, then Lemma A.7 follows immediately from the congruence of *addcontext* and *addcontext$_A$* (Lemma A.4). The extended f maps the new context in σ to the new context in σ_A.

If *exp* is a **new**(*type*) instruction, then let $(\sigma', fl', label', loc') = addstruct\,(\sigma, fl, label, ptocc, type)$, and $(\sigma'_A, fl'_A, label'_A, loc'_A) = addstruct_A\,(\sigma_A, fl_A, label_A, ptocc', type')$. By the congruence of *addstruct* and *addstruct$_A$* (Lemma A.5), $(\sigma, label) \rhd (\sigma', label')$ by an extended f that sends loc' to loc'_A. If *type* = **env**, then Lemma A.7 is immediate. Otherwise, the proof of the **new** case is completed by using the congruence of *addatom* and *addatom$_A$* (Lemma A.6) to argue that adding a set of new references at loc' and loc'_A to new **nil** atoms preserves the embedding from the updated $(\sigma', label')$ to the updated $(\sigma'_A, label'_A)$.

Finally, if *exp* is the primitive operator $op(exp_1, \cdots, exp_n)$, then the definition of $\mathcal{H}$ ensures that each of the exp_i's are either atoms, or identifier expressions that denote atoms. An induction on n can be used to show that if $simplexp(state, exp)$ invokes *op* on a set of atoms $loc_1 \cdots loc_n$ then $simplexp_A(state_A, exp)$ invokes *op* on a set of atoms $loc'_1 \cdots loc'_n$, where each of the loc'_i has a value that subsumes the value of the corresponding loc_1. Lemma A.7 then follows from the assumption that the approximation semantics' primitive operands are isotone in their arguments. □

LEMMA A.8 (***congruence of cond, cond$_T$, cond$_F$***). Let $\sigma \rhd \sigma_A$ by f. Then $cond(pt, \sigma, cexp)$ implies $cond_T(pt, \sigma_A, cexp)$, and $\neg cond(pt, \sigma, cexp)$ implies $cond_F(pt, \sigma_A, cexp)$.

PROOF. This lemma is proved on the depth to which *not* operators are nested in *cexp*.

BASIS (*cexp* contains no negations). The proof is by an analysis of the type of *cexp*.

If *cexp* is the predicate $typeOf(exp, type)$ expression, then let $loc = idexpr(pt, \sigma, exp)$ and $locs' = idexpr_A(pt, \sigma_A, exp)$. By the congruence of *idexpr* and $idexpr_A$ (Lemma A.3), $f_{struct}(loc) \in locs'$. If $gettype(\sigma, loc) = type$, then the congruence of *gettype* and $gettype_A$ (Lemma A.2) implies that $type \in gettype_A(\sigma, f_{struct}(loc))$. Then, by the definition of *maybe_oftype*, $cond_T(pt, \sigma_A, cexp)$ is true. Similarly, if $gettype(\sigma, loc) = type'$, where $type' \neq type$, then the congruence of *gettype* and $gettype_A$ implies that $type' \in gettype_A(\sigma, f_{struct}(loc))$. Then, by the definition of *maybe_not_oftype*, $cond_F(pt, \sigma_A, cexp)$ is true.

If *cexp* is the predicate $Eq(exp_1, exp_2)$, then let $loc_1 = idexpr(pt, \sigma, exp_1)$, $loc_2 = idexpr(pt, \sigma, exp_2)$, $locs'_1 = idexpr_A(pt, \sigma_A, exp_1)$, and $locs'_2 = idexpr_A(pt, \sigma_A, exp_2)$. Then, by the congruence of *idexpr* and $idexpr_A$ (Lemma A.3), $f_{struct}(loc_1) \in locs'_1$ and $f_{struct}(loc_2) \in locs'_2$. If $loc_1 = loc_2$, then the fact that f is a map implies that $f_{struct}(loc_1) = f_{struct}(loc_2)$. By the definition of *may_coincide*, $cond_T(pt, \sigma_A, cexp)$ is true. Otherwise, if $loc_1 \neq loc_2$, then f either sends loc_1 and loc_2 to the different locations, or to the same location. In the former case, $f_{struct}(loc_1) \neq f_{struct}(loc_2)$, and the definition of *may_differ* implies $cond_F(pt, \sigma_A, cexp)$ is true. Otherwise, $f_{struct}(loc_1)$, by the definition of an embedding, must be a summary structure. The definition of *may_differ* again implies $cond_F(pt, \sigma_A, cexp)$ is true.

The argument that the base case of Lemma A.8 is true when *cexp* is a comparison operator is similar to the argument that primitive operators are isotone (*cf.* Lemma A.7).

INDUCTION HYPOTHESIS. The lemma holds when *cexp* contains k negations.

INDUCTION STEP. Let *cexp* be of the form *not*(*cexp'*). Then the assertion that $cond(pt, \sigma, cexp)$ implies $cond_T(pt, \sigma_A, cexp)$, by the definition of $cond_T$, is equivalent to the assertion that $\neg cond(pt, \sigma, cexp')$ implies $cond_F(pt, \sigma_A, cexp')$. Lemma A.8 now follows from the induction hypothesis. A similar argument shows that Lemma A.8 holds for $cond_F$ when *cexp* is nested to depth $k+1$. □

LEMMA A.9 (***congruence of evalPt$_I$ and evalPt$_A$***). Let $state = (pt, \sigma, fl, label, occ)$ and $state_A = (pt, \sigma_A, fl_A, label_A, occ_A)$ be states such that $state \rhd state_A$ w.r.t. an f that maps σ onto σ_A. Then $evalPt_I(state, exp)$ and $result'_* = evalPt_A(state', exp)$. Then there exists a $(\sigma'_A, fl'_A, label'_A)$ in $result'_*$ such that $(\sigma', label') \rhd (\sigma'_A, label'_A)$.

PROOF. The proof is by an analysis of the type of *pt*.

If *pt* is the predicate *If* (*cexp*), let $bool = cond\,(pt, \sigma, cexp)$, $bool_T = cond_T\,(pt, \sigma_A, cexp)$, and $bool_F = cond_F\,(pt, \sigma_A, cexp)$. By the congruence of *cond*, $cond_T$, and $cond_F$ (Lemma A.8), *bool* implies $bool_T$ and $\neg bool$ implies $bool_F$. Lemma A.9 now follows immediately from the definition of $evalPt_I$.

The proof of Lemma A.9 for when *pt* is the predicate *While* (*cexp*) is similar.

If *pt* is the statement *Assign* (*lexp.sel*, *rexp*), then let $(\sigma_r, fl_r, label_r, rvloc) = simplexp\,(pt, state, rexp)$ and $rresult'_* = simplexp_A\,(pt, state', rexp)$. By the congruence of *simplexp* and $simplexp_A$ (Lemma A.7), $rresult'_*$ contains a $(\sigma'_r, fl'_r, label'_r, rvloc')$ such that $(\sigma_r, label_r) \rhd (\sigma'_r, label'_r)$ by an f' that maps *rvloc* to *rvloc'*.

Let $lvloc = idexpr\,(pt, \sigma_r, lexp)$ and $lresult'_* = idexpr_A\,(pt, \sigma_{A_r}, lexp)$. Then, by the congruence of *idexpr* and $idexpr_A$ (Lemma A.3), $f'_{struct}\,(lvloc) \in lresult'_*$.

Let $(\sigma', label')$ denote the effect of setting reference *sel* at *lvloc* in σ to *rvloc*, and updating *label* accordingly. Let $assign'_* = updref(\sigma_{A_r}, fl_{A_r}, label_{A_r}, ptocc', f'(lvloc), sel, rvloc')$. The definition of *updref* now implies that there is a $(\sigma'_A, fl'_A, label'_A)$ in $assign'_*$ that abstracts $(\sigma', label')$.

Lemma A.9 now follows from the observation that $evalPt_I\,(state)$ returns $(nextpt, \sigma', fl', label', occ)$ only if $evalPt_A\,(state)$ returns $(nextpt, \sigma'_A, fl'_A, label'_A, occ_A)$.

If *pt* is a **call** statement, then Lemma A.9 follows immediately from the definition of $evalPt_I$.

If *pt* is a **return** statement, let $cloc = idexpr\,(pt, \sigma, \_curr.\_callctxt)$ and $ctxloc'_* = idexpr\,(pt, \sigma_A, \_curr.\_callctxt)$. Then, by the congruence of *idexpr* and $idexpr_A$, $f_{struct}\,(cloc) \in ctxloc'_*$.

Let $(retpt, retocc) = getcxt\,(\sigma, cloc)$ and $ctxloc'_* = getcxt\,(\sigma_A, f_{struct}\,(cloc))$. Let $(retpt, apxocc_A)$ be an arbitrary element of $ctxloc_*$. By the congruence of *getcxt* and $getcxt_A$ (Lemma A.1), $ctxloc'_*$ contains a $(retpt, apxocc')$ such that $retocc \rhd apxocc_A$.

Let $prevenv_* = idexpr\,(pt, \sigma, \_curr.\_callctxt)$ and $prevenv'_* = idexpr\,(pt, \sigma_A, \_curr.\_callctxt)$. Let *eloc* be an arbitrary element of $prevenv_*$. Then, by the congruence of *idexpr* and $idexpr_A$, $f_{struct}\,(eloc) \in prevenv'_*$.

By the definition of abstraction, *eloc*'s creation-point-label is abstracted by $f_{struct}\,(eloc)$'s label. Then $f_{struct}\,(eloc)$ must be consistent with *retpt*.

Let *gEnv* denote σ's global environment. By the definition of abstraction, the global environment of σ_A is $f_{struct}\,(gEnv)$. Let $(\sigma_r, label_r)$ denote the effect of resetting the local environment of σ to *eloc*. Let $new'_* = updref\,(\sigma_A, label_A, ptocc', f_{struct}\,(gEnv), \_curr, f_{struct}\,(eloc))$. By the definition of *updref*, new'_* contains a $(\sigma'_r, label'_r)$ such that $(\sigma_r, label_r) \rhd (\sigma'_r, label'_r)$. Lemma A.9 now follows immediately from the observation that $evalPt_I\,(state)$ returns $(retpt, \sigma_r, fl, label_r, retocc)$ iff $evalPt_A\,(state')$ returns $(retpt, \sigma'_r, fl_A, label'_r, apxocc_A)$.

Finally, if *pt* is an initialization statement, then Lemma A.9 follows immediately from the definition of $evalPt_I$ and $evalPt_A$. □

Appendix 7. A Semantics for the Language $\mathcal{S}$

The following is a formal semantics for the language $\mathcal{S}$.

$State = Store \times Freelist \quad Store = Var \to Value \quad Freelist = Ref^* \quad Value = Atom + Ref$

$\mathbf{M}_{\mathcal{S}} : Prog \to Store \to Freelist \to Store_\perp$
$\mathbf{M}_{\mathcal{S}}(prog, \sigma, fl) =$
 let $program = finalize(initialize(prog))$ **in**
  let $evalPgm = fix\ \lambda f.\lambda((pt, \sigma', fl)).\ pt \stackrel{?}{=} \mathbf{final} \to \sigma' \,[]\, f(evalPt_{\mathcal{S}}((pt, \sigma', fl)))$
  in $evalPgm((\mathbf{initial}_1, \sigma, fl))$
 end*

Function *fix* is the least fixpoint functional.

The function *initialize*(*prog*) prepends a two-part prologue to *prog*. The first part of this prologue is the statement "[**initial**$_1$] *initialize* ;". The second part simulates the call to *main*( ) at point "[**initial**$_2$]"; *i.e.*, it simulates the allocation of *main*( )'s local environment, and subsequent transfer of control to *main*( )'s first statement.

The function *finalize*(*prog*) appends a two-part epilogue to *prog*. The first part of this epilogue simulates the implicit **return** statement at the end of *main*( ); *i.e.*, it resets the local environment, and transfers control to point **final**. The second part of this epilogue is the statement "[**final**] *skip*".

$evalPt_{\mathcal{S}} : State \to State_\perp$
$evalPt_{\mathcal{S}}((pt, \sigma, fl))$
 case *formOf* (*pt*, *program*) **in**

If (*cexp*):	**let** $nextpt_T$ and $nextpt_F$ be *pt*'s true and false control-flow successors **in** $(cond(pt, \sigma, cexp) \to nextpt_T \,[]\, nextpt_F, \sigma, fl)$ **end**
Case (*exp*, *cases*):	**let** $(fl', value) = rvalue(\sigma, fl, exp)$ **in** **let** $evalCase = fix\ \lambda f.\lambda cases'.$ $cases' \stackrel{?}{=} \varepsilon \to \perp$ [] **let** $((cguard, cpoint), cases'') = cases'$ **in** $value \stackrel{?}{=} cguard \to (cpoint, \sigma, fl')\ []\ f(cases'')$ **end** **in** $evalCase(cases)$ **end***
Assign (*lexp*, *rexp*):	**let** *nextpt* be *pt* 's control-flow successor **in** **let** $(fl', value) = rvalue(\sigma, fl, exp)$ **in** $(nextpt, \sigma[value / var], fl')$ **end** **end**
Assert (*cexp*):	**let** *nextpt* be *pt* 's control-flow successor **in** $cond(\sigma, cexp) \to (nextpt, \sigma, fl)$ [] $\perp$ **end**
Initialize:	**let** *nextpt* be *pt* 's control-flow successor **in** **let** σ' be a copy of σ **in** $(nextpt, \sigma', fl)$ **end** **end**
Fail:	$\perp$

end

Function *formOf* pairs every program point with its associated syntactic construct.

$cond : Store \times Cond \rightarrow Bool_\perp$
$cond\ (\sigma, cexp) =$
case *cexp* **in**

$TypeOf\ (var, type)$:	$var \stackrel{?}{=} type$
$Compare\ (exp_1, op, exp_2)$:	**let** $val_1 = simplexp\ (\sigma, exp_1)$ **and** $val_2 = simplexp\ (\sigma, exp_2)$ **in** $\{ val_1, val_2 \} \subseteq Atom \rightarrow (val_1\ op\ val_2)\ [] \perp$ **end**
$Not\ (cexp)$:	$\neg\ cond\ (\sigma, cexp)$

esac

$rvalue : Store \times Freelist \times Exp \rightarrow (Freelist \times Value)_\perp$
$rvalue\ (\sigma, fl, exp) =$
case *exp* **in**

$Primop\ (op, exp_1, \cdots, exp_n)$:	$(fl, op(simplexp\ (\sigma, sexp_1), \cdots, simplexp\ (\sigma, sexp_n))$
$Freelist\ ()$:	$read\ (fl)$
$Atom\ (exp), Var\ (exp)$:	$(fl, simplexp\ (\sigma, exp))$

esac

$simplexp : Store \times Exp \rightarrow Value = \lambda\ (\sigma, exp)\ .\ exp \in Atom \rightarrow exp\ [] \ \sigma(exp)$

The variable *program* is treated as a global to simplify the semantics. This variable is implicitly referenced by statements that determine a program point's control-flow successors.

The function $read : Freelist \rightarrow (Ref \times Freelist)_\perp$ accepts one argument, a stream s. It returns $\perp$ if s is empty, and the first element of s paired with the tail of s otherwise.

Appendix 8. Definition of an *Spdg*

An *S-language pdg* (*spdg*) gives a control-flow-graph-based characterization of a program's control dependences, and an execution-based characterization of its flow and def-order dependences. The following is the formal definition of an *spdg*.

A program P_S's *spdg* must contain one edge for each of P_S's static control dependences (*cf.* §3.2.2.1). Edges that correspond to true control dependences are labeled **true**; edges that correspond to false control dependences are labeled **false**.

A program's data dependences are defined with the aid of a state transition relation, $\vdash_S$. This relation is similar to state transition relations defined in Chapters 3 and 4: *i.e.*,

DEFINITION. A ***state*** in a language S computation is a (program-point,store,freelist) triple. □

DEFINITION. The ***state transition relation*** $\cdots \vdash_S \cdots \longrightarrow \cdots$ is defined as follows:

$$prog \vdash_S state_i \longrightarrow^0 state_j \Leftrightarrow state_j = state_i$$

$$prog \vdash_S state_i \longrightarrow^n state_j \Leftrightarrow \exists\, state' : prog \vdash_S state_i \longrightarrow^{n-1} state' \wedge state_j = evalPt(prog, state')$$

$$prog \vdash_S state_i \longrightarrow^* state_j \Leftrightarrow \exists\, n : prog \vdash_S state_i \longrightarrow^n state_j$$

$$prog \vdash_S state_i \longrightarrow^+ state_j \Leftrightarrow \exists\, n > 0 : prog \vdash_S state_i \longrightarrow^n state_j$$

$$prog \vdash_S state_n \longrightarrow \cdots \longrightarrow state_m \Leftrightarrow \forall\, i : n \le i \le m-1 : prog \vdash_S state_i \longrightarrow^1 state_{i+1}$$ □

The expression $evalPt_S(prog, state')$ (once again) constitutes a minor abuse of notation. The function $evalPt_S$ (*cf.* Appendix 7) actually takes one formal parameter—a state—and four non-local parameters that describe *prog*'s control-flow graph, structure declarations, and local identifiers. The definitions of data dependence are now similar to the ones given in Chapter 3.

P_S's *spdg* w.r.t. *InSet* must depict $p \longrightarrow_f q$ whenever P_S exhibits $p \longrightarrow_f q$ w.r.t. *InSet*. Similarly, P_S's *spdg* w.r.t. *InSet* must depict $p \longrightarrow_{do(r)} q$ whenever P_S exhibits $p \longrightarrow_{do(r)} q$ w.r.t. *InSet*. Labels that identify loop-carried and loop-independent dependences are not needed in the *spdg*, since S is a loop-free language.

Program P_S's *spdg* must also contain one edge for each of P_S's stream-mediated data dependences. Stream-mediated dependences, however, play only a minor role in the thesis; the proof of the Simulation Equivalence Lemma reduces programs that access the simulated freelist to equivalent programs that do not access the freelist.

REFERENCES

Abr87.

Abramsky, S. ed. and Hankin, C. ed., *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited, Chichester, West Sussex, England (1987).

Aho86.

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

All83.

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).

All83a.

Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J., "Conversion of Control Dependence to Data Dependence," pp. 177-189 in *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 24-26, 1983), ACM, New York, NY (1983).

All84.

Allen, J.R. and Kennedy, K., "Automatic Loop Interchange," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, (Montreal, Can., June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 233-246 (June 1984).

All86.

Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A., "PTOOL: A Semi-automatic Parallel Programming Assistant," Tech Rep. COMP TR86-31, Dept. of Computer Science, Rice Univ., Houston, TX (January 1986).

All87.

Allen, R. and Kennedy, K., "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems* **9**(4) pp. 491-542 (October 1987).

All88.

Allen, R. and Johnson, S., "Compiling C for Vectorization, Parallelization, and Inline Expansion," *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. 241-249 (July 1988).

Alp88.

Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 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).

Ame89.

American National Standards Institute, Inc.,, "Fortran 8x Draft: May 1989," *FORTRAN Forum: ACM SIGPLAN special interest publication on FORTRAN* **8**(4)(December 1989).

Bac78.

Backus, J., "Can Programming Languages be Liberated from the Von Neumann Style? A Functional

Style and an Algebra of Programs," *Comm. of the ACM* **21**(8) pp. 613-641 (August 1978).

Bal89.

Balasundram, V., "Interactive Parallelization of Numerical Scientific Programs," Ph.D. dissertation and Tech Rep. COMP TR89-95, Dept. of Computer Science, Rice Univ., Houston, TX (July, 1989).

Bal89a.

Balasundram, V. and Kennedy, K., "Compile-Time Detection of Race Conditions in a Parallel Program," pp. 175-185 in *Proceedings of the Third International Conference on Supercomputing*, Crete, Greece (June 1989).

Bal90.

Ballance, R.A., Maccabe, A.B., and Ottenstein, K.J., "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 257-271 (June 1990).

Bax89.

Baxter, W. and Bauer, III, H.R., "The Program Dependence Graph and Vectorization," pp. 1-11 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

Ber66.

Bernstein, A.J., "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers* **15**(5) pp. 757-763 (October 1966).

Bin89.

Binkley, D., Horwitz, S., and Reps, T., "The Multi-Procedure Equivalence Theorem," TR-890, Computer Sciences Department, University of Wisconsin, Madison, WI (November 1989).

Bin91.

Binkley, D., "Multi-Procedure Program Integration," pending (thesis), Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

Bod90.

Bodin, F., "Preliminary Report: Data Structure Analysis in C Programs," *Proceedings of the Workshop on Parallelism in the Presence of Pointers and Dynamically-allocated Objects* (Leesburg, Virginia, March 1990), *Technical Note SRC-TN-90-292*, pp. 4.3.1-4.3.34 Supercomputing Research Center/Institute for Defense Analysis, (1990).

Cal87.

Callahan, D., "A Global Approach to Detection of Parallelism," Ph.D. dissertation and Tech Rep. COMP TR87-50, Dept. of Computer Science, Rice Univ., Houston, TX (April 1987).

Car89.

Cartwright, R. and Felleisen, M., "The Semantics of Program Dependence," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 13-27 (July 1989).

Car91.

Cartwright, R. and Fagan, M., "Soft Typing," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto, Ontario, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 278-292 (June 1991).

Cha87.

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

Cha88.

Chase, D.R., "Safety Considerations for Storage Allocation Optimization," *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. 1-10 (July 1988).

Cha90.

Chase, D.R., Wegman, M., and Zadeck, F.K., "Analysis of Pointers and Structures," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6)(June 1990).

Cle88.

Cleary, T., (Translator's Introduction to Sun Tzu's) *The Art of War*, Shambhalla, Boston, MA (1988).

Cou77.

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).

Cou78.

Cousot, P., "Méthodes Iteratives de Construction et d'Approximation de Pointes Fixes d'Operateurs Monotones sur un Trellis, Analyse Sémantique des Programmes," Ph.D. dissertation, Université Scientifique et Médicale de Grenoble/Institut National Polytechnique de Grenoble (March 1978).

Cou80.

Cousot, P. and Cousot, R., "Systematic Design of Program Analysis Frameworks," pp. 70-85 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1980).

Cou86.

Coutant, D., "Retargetable High-Level Alias Analysis," pp. 157-168 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).

Cyt86.

Cytron, R., Lowry, A., and Zadeck, K., "Code Motion of Control Structures in High-Level Languages," pp. 70-85 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).

Deb89.

Débray, S., "Static Inference of Modes and Data Dependences in Logic Programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 418-450 (July 1989).

Deu90.

Deutsch, A., "On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-order Functional Specifications," pp. 157-168 in *Conference Record of the 17th ACM Symposium on Principles of Programming Languages*, (San Francisco, CA, Jan. 17-19, 1990), ACM, New York, NY (1990).

Die87.

Dietz, H., *The Refined-Language Approach to Compiling for Parallel Supercomputers*, UMI Research Press, Ann Arbor, MI (May 1987).

Fer83.

Ferrante, J. and Ottenstein, K., "A Program Form Based on Data Dependency in Predicate Regions," pp. 217-236 in *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 24-26, 1983), ACM, New York, NY (1983).

Fer83a.

Ferrante, J., Ottenstein, K., and Warren, J., "The Program Dependence Graph and Its Use in Optimization," Res. Rep. RC10208, IBM T.J. Watson Research Center, Yorktown Heights, NY (August 1983).

Fer87.

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).

Ger75.

Gerhart, S.L., "Correctness-Preserving Program Transformations," pp. 54-66 in *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, (Palo Alto, CA, Jan. 20-22, 1975), ACM, New York, NY (1975).

Goh90.

Gohkale, M. and Smith, L., "Alias Analysis of Recursively Defined Structures," *Proceedings of the Workshop on Parallelism in the Presence of Pointers and Dynamically-allocated Objects* (Leesburg, Virginia, March 1990), *Technical Note SRC-TN-90-292*, pp. 5.8.1-5.8.19 Supercomputing Research Center/Institute for Defense Analysis, (1990).

Gon69.

Gonzalez, M.J. and Ramamoorthy, C.V., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," pp. 1-15 in *AFIPS Fall Joint Computer Conference* (Las Vegas, Nevada, November 18-20, 1969), (1969).

Gua90.

Guarna Jr., V.A., "Dependence Analysis for C Programs Containing Pointers and Dynamic Data Structures," *Proceedings of the Workshop on Parallelism in the Presence of Pointers and Dynamically-allocated Objects* (Leesburg, Virginia, March 1990), *Technical Note SRC-TN-90-292*,

pp. 5.15.1-5.15.25 Supercomputing Research Center/Institute for Defense Analysis, (1990).

Gua90a.

Guarna Jr., V.A., "Symbolic Alias and Side Effect Analysis of Pointers and Dynamic Memory Structures for the Parallelization of C," Preprint, Motorola Microcomputer Division, Urbana Design Center (October, 1990).

Hab86.

Habel, A. and Kreowski, H-J., "May We Introduce You to Hyperedge Replacement," pp. 15-26 in *Proceedings of the Third International Workshop on Graph-Grammars and Their Application to Computer Science*, (Warrenton, Virginia, December 1986), *Lecture Notes in Computer Science* **291**, ed. H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld,Springer-Verlag, New York, NY (1986).

Har89.

Harrison, W.L., "The Interprocedural Analysis and Automatic Parallelization of Scheme Programs," Ph.D. Thesis, CSRD Tech. Rep. 860, Center for supercomputing research and development, University of Illinois at Urbana-Champaign, Urbana, Ill. (February 1989).

Hed88.

Hederman, L., "Compile Time Garbage Collection Using Reference Count Analysis," M.Sc. dissertation and Tech. Rep. COMP TR88-75, Dept. of Computer Science, Rice Univ., Houston, TX (August 1988).

Hen89.

Hendren, L. and Nicolau, A., "Interference Analysis Tools for Parallelizing Programs with Recursive Data Structures," in *Proceedings of the Third International Conference on Supercomputing*, Crete, Greece (June 1989).

Hen90.

Hendren, L., "Parallelizing Programs with Recursive Data Structures," Ph.D. Thesis, Tech. Rep. 90-1114, Dept. of Computer Science, Cornell University, Ithaca, NY (April 1990).

Hoa78.

Hoare, C.A.R., "Some Properties of Predicate Transformers," *Journal of the ACM* **25**(3) pp. 461-480 (July 1978).

Hoo86.

Hood, R., Kennedy, K., and Müller, H.A., "Efficient Recompilation of Module Interfaces in a Software Development Environment," Technical Report DCS-59-IR, Dept. of Computer Science, Univ. of Victoria, Victoria, B.C. (1986).

Hor87.

Horwitz, S., Prins, J., and Reps, T., "Integrating Non-interfering Versions of Programs," TR-690, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1987).

Hor87a.

Horwitz, S., Prins, J., and Reps, T., "On the Adequacy of Program Dependence Graphs for Representing Programs," TR-699, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1987).

Hor88.

Horwitz, S., Prins, J., and Reps, T., "On the Adequacy of Program Dependence Graphs for Representing Programs," pp. 146-157 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).

Hor88a.

Horwitz, S., Reps, T., and Binkley, D., "Interprocedural Slicing Using Dependence Graphs," TR-756, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1988).

Hor89a.

Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence Analysis for Pointer Variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).

Hor89.

Horwitz, S., Prins, J., and Reps, T., "Integrating Non-interfering Versions of Programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).

Hor90.

Horwitz, S., "Identifying the Semantic and Textual Differences Between Two Versions of a Program," *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, (White Plains, NY, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6)(July 1990).

Hor90a.

Horwitz, S., Reps, T., and Binkley, D., "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).

Hud86.

Hudak, P., "A Semantic Model of Reference Counting and its Abstraction (Detailed Summary)," pp. 351-363 in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, (Cambridge, MA, August 4-6, 1986), ACM, New York, NY (1986).

Hud87.

Hudak, P., "A Semantic Model of Reference Counting and its Abstraction ," pp. 45-62 in *Abstract Interpretation of Declarative Languages*, Ellis Horwood Limited, Chichester, West Sussex, England (1987).

Hud88.

Hudak, P., "Exploring Parafunctional Programming: Separating the What from the How," *IEEE Software*, pp. 54-61 (January 1988).

Hud91.

Hudak, P. and Young, J., "Collecting Interpretations of Expressions ," *ACM Transactions on Programming Languages and Systems* **13**(2) pp. 269-290 (April 1991).

Hun76.

Hunt, J.W. and McIlroy, M.D., "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. 41, Bell Laboratories, Murray Hill, NJ (1976).

Hwa88.

Hwang, J.C., Du, M.W., and Chou, C.R., "Finding Program Slices for Recursive Procedures," in *Proceedings of IEEE COMPSAC 88*, (Chicago, IL, Oct. 3-7, 1988), IEEE Computer Society, Washington, DC (1988).

Ino88.

Inoue, K., Seki, H., and Yagi, H., "Analysis of Functional Programs to Detect Run-time Garbage Cells," *ACM Transactions on Programming Languages and Systems* **10**(4) pp. 555-578 (October 1988).

Jon79.

Jones, N.D. and Muchnick, S.S., "Flow Analysis and Optimization of Lisp-like Structures," pp. 244-256 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).

Jon81.

Jones, N.D. and Muchnick, S.S., "Flow Analysis and Optimization of Lisp-like Structures," pp. 102-131 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

Jon82.

Jones, N.D. and Muchnick, S.S., "A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures," pp. 66-74 in *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, (Albuquerque, NM, January 25-27, 1982), ACM, New York, NY (1982).

Jon86.

Jones, N.D. and Mycroft, A., "Data Flow Analysis of Applicative Programs Using Minimal Function Graphs: Abridged Version," pp. 296-306 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, (St. Petersburg, FL, Jan 13-15, 1986), ACM, New York, NY (1986).

Jon88.

Jones, Neil D., "Challenging Problems in Partial Evaluation and Mixed Computation," pp. 1-14 in *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation* (Gammel Avernaes, Denmark, 18-24 October, 1987), ed. D. Bjorner, A.P. Ershov, N. D. Jones,Elsevier (North-Holland) (1988).

Jor86.

Jorring, U. and Scherlis, W., "Deriving and Using Destructive Data Types," in *TC 2 Working Conference on Program Specification and Transformation*, (Bad Tolz, West Germany, April 15-17, 1986), (March 1986).

Jou91.

Jouvelot, P. and Gifford, D., "Algebraic Reconstruction of Types and Effects," pp. 303-309 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 21-23, 1991), ACM, New York, NY (1991).

Kam76.

Kam, J.B. and Ullman, J.D., "Global Dataflow Analysis and Iterative Frameworks," *Journal of the ACM* **23**(1) pp. 158-171 (January 1976).

Kau63.

Kaufmann, W., *The Faith of a Heretic*, Anchor Books, United States (1963).

Kil73.

Kildall, G., "A Unified Approach to Global Program Optimization," pp. 194-206 in *Conference Record of the ACM Symposium on Principles of Programming Languages*, (Boston, MA, October 1-3, 1973), ACM, New York, NY (1973).

Kuc72.

Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).

Kuc78.

Kuck, D.J., *The Structure of Computers and Computations, Vol. 1*, John Wiley and Sons, New York, NY (1978).

Kuc81.

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).

Kuh80.

Kuhn, R. H., "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. dissertation and Tech. Rep. R-80-1009, Dept. of Computer Science, University of Illinois, Urbana, IL (February 1980).

Lam74.

Lamport, L., "The Parallel Execution of DO Loops," *Comm. of the ACM* **17**(2) pp. 83-93 (February 1974).

Lan90.

Landi, W. and Ryder, B.G., "Aliasing with and without Pointers: A Problem Taxonomy," Tech. Rep. CAIP-TR-255, Rutgers Univ., New Brunswick, NJ (September 1990).

Lan91.

Landi, W. and Ryder, B.G., "Pointer-Induced Aliasing: A Problem Taxonomy," pp. 47-57 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 21-23, 1991), ACM, New York, NY (1991).

Lar87.

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).

Lar88.

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).

Lar89.

Larus, J.R., "Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors," Ph.D. dissertation and Tech. Rep. UCB/CSD 89/502, Computer Science Division, Dept. of Elec. Eng. and Comp. Sci., Univ. of California – Berkeley, Berkeley, CA (May 1989).

Man74.

Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, New York, NY (1974).

Mas86.

Mason, I., *The Semantics of Destructive Lisp*, Center for the Study of Language and Information, Menlo Park, CA (1986).

Mas90.

Mason, I. and Talcott, C., "Reasoning about Programs with Effects," pp. 189-203 in *Programming Language Implementation and Logic Programming: Proceedings of an International Workshop*, (Linköping, Sweden, August 1990), *Lecture Notes in Computer Science* **456**, ed. P. Deransart, T. Maluszyński,Springer-Verlag, New York, NY (1990).

Mil88.

Miller, B. and Choi, J.-D., "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, GA, June. 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 135-144 (July 1988).

Mil76.

Milne, R. and Strachey, C., *A Theory of Programming Language Semantics*, Halsted Press, New York, NY (1976).

Mul87.

Mulmuley, K., *Full Abstraction and Semantic Equivalence*, The M.I.T. Press, Cambridge, MA (1987).

Mur71.

Muraoka, Y., "Parallelism Exposure and Exploitation in Programs," Ph.D. dissertation and Tech. Rep. R-71-424, Dept. of Computer Science, Univ. of Illinois, Urbana, IL (February 1971).

Myc81.

Mycroft, A., "Abstract Interpretation and Optimising Transformations for Applicative Programs," Ph.D. dissertation and Tech. Rep. CST-15-81, University of Edinburgh, Edinburgh, U.K. (December 1981).

Myc83.

Mycroft, A. and Nielson, F., "Strong Abstract Interpretation Using Power Domains," pp. 192-205 in *10th Colloquium on Automata, Languages, and Programming*, (Barcelona, Spain, July 18-22, 1983), *Lecture Notes in Computer Science* **154**, ed. H. Ganzinger and N.D. Jones,Springer-Verlag, New York, NY (1983).

Myc85.

Mycroft, A. and Jones, N.D., "A Relational Framework for Abstract Interpretation," pp. 156-171 in *Proceedings of a Workshop on Programs as Data Objects*, (Copenhagen, DK, October 17-19, 1985), *Lecture Notes in Computer Science* **217**, ed. H. Ganzinger and N.D. Jones,Springer-Verlag, New York, NY (1985).

Mye81.

Myers, E., "A Precise Inter-procedural Data Flow Algorithm," pp. 219-230 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

Nag79.

Nagl, M., "A Tutorial and Bibliographical Survey on Graph Grammars," pp. 70-126 in *Graph-grammars and Their Application to Computer Science and Biology*, (Bad Honnef, October 30-November 3, 1978), *Lecture Notes in Computer Science* **73**, ed. V. Claus, H. Ehrig, and G. Rozenberg,Springer-Verlag, New York, NY (1979).

Nei88.

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).

Net91a.

Netzer, Robert H. B. and Miller, Barton P., "Detecting Data Races in Parallel Program Executions," in *Languages and Compilers for Parallel Computing*, ed. D. Gelernter, T. Gross, A. Nicolau, and D. Padua,MIT Press (1991). Also appears in the *3rd Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, (Aug. 1990).

Net91.

Netzer, Robert H. B. and Miller, Barton P., "Improving the Accuracy of Data Race Detection," *3rd ACM Symposium on Principles and Practice of Parallel Programming*, pp. 133-144 Williamsburg, VA, (April 1991).

Nie81.

Nielson, F., "Semantic Foundations of Dataflow Analysis," M. Sc. Thesis, Tech. Rep. DAIMI PB-131, Computer Science Dept., Aarhus Univ., Aarhus, Denmark (1981).

Nie84.

Nielson, F., "Abstract Interpretation Using Domain Theory," Ph.D. dissertation and Tech. Rep. CST-31-84, University of Edinburgh, Edinburgh, U.K. (October, 1984).

Nie87.

Nielson, F., "Towards a denotational theory of abstract interpretation," pp. 219-245 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky and C. Hankin,Ellis Horwood Limited, Chichester, West Sussex, England (1987).

Nie90.

Nielson, F., "Theoretical Aspects of Semantics-based Language Implementation," D. Sc. Thesis, Tech. Rep. DAIMI PB-329, Computer Science Dept., Aarhus Univ., Aarhus, Denmark (August

1990).

Oat77.

Oates, S.B., *With Malice Towards None: The Life of Abraham Lincoln*, N.A.L. Penguin, Inc., New York, NY (1977).

Ott78.

Ottenstein, K.J., *Data-Flow Graphs as an Intermediate Program Form*, UMI Research Press, Ann Arbor, MI (August 1978).

Ott84.

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).

Pad79.

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).

Par83.

Partsch, H. and Steinbrüggen, R., "Program Transformation Systems," *ACM Computing Surveys* **15**(3) pp. 199-236 (September 1983).

Per82.

Perlis, A.J., "Epigrams on Programming," *ACM SIGPLAN Notices* **17**(9) pp. 7-13 (September 1982).

Pfe91.

Pfeiffer, P. and Selke, R., "On the Adequacy of Dependence-based Representations for Programs with Heaps," TR-992, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1991).

Pfe91a.

Pfeiffer, P. and Selke, R., "On the Adequacy of Dependence-based Representations for Programs with Heaps," in *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, (Sendai, Japan, September 24-27, 1991), Springer-Verlag, New York, NY (1991).

Pin91.

Pingali, K., Beck, M., Johnson, R., Moudgill, M., and Stodghill, P., "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies," pp. 67-78 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 21-23, 1991), ACM, New York, NY (1991).

Ple81.

Pleban, U., "Preexecution Analysis Based on Denotational Semantics," Ph.D. dissertation , Computer Sciences Dept., Univ. of Kansas (1981).

Pod90.

Podgurski, A. and Clarke, L., "A Formal Model of Program Dependences and Its Implications for

Software Testing, Debugging, and Maintenance," *IEEE Trans. on Software Eng.* **16**(9) pp. 965-978 (September 1990).

Ram89.

Ramalingam, G. and Reps, T., "Semantics of program representation graphs," TR-900, Computer Sciences Department, University of Wisconsin, Madison, WI (December 1989).

Rei81.

Reif, J.H. and Tarjan, R.E., "Symbolic Program Analysis in Almost Linear Time," *SIAM J. of Comput.* **11**(1) pp. 81-93 (February 1981).

Rep89.

Reps, T. and Yang, W., "The Semantics of Program Slicing and Program Integration," *Proceedings of the International Joint Conference on Theory and Practice of Software Development (Colloquium on Current Issues in Programming Languages)*, (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science* **352** pp. 360-374 Springer-Verlag, (1989).

Rey68.

Reynolds, J.C., "Automatic Computation of Data Set Definitions," pp. 456-461 in *Information Processing 68: Proceedings of the IFIP Congress 68*, North-Holland, New York, NY (1968).

Rug87.

Ruggieri, C., "Dynamic Memory Allocation Techniques Based on the Lifetime of Objects," Ph.D. dissertation, Purdue University, UMI Research Press, Ann Arbor, MI (August 1987).

Rug88.

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).

Sag90.

Sagiv, S., Francez, N., Rodeh, M., and Wilhelm, R., "A Logic-Based Approach to Data Flow Analysis Problems (*Preliminary Version*)," pp. 277-292 in *Programming Language Implementation and Logic Programming: Proceedings of an International Workshop*, (Linköping, Sweden, August 1990), *Lecture Notes in Computer Science* **456**, ed. P. Deransart, T. Maluszyński,Springer-Verlag, New York, NY (1990).

Sch75.

Schwartz, J.T., "Optimization of very high level languages," *Computer Languages* **1** pp. 161-218 (1975).

Sel89.

Selke, R.P., "A Rewriting Semantics for Program Dependence Graphs," pp. 12-24 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

Sel90.

Selke, R.P., "Program Dependence Graphs: A Formal Treatment," Technical Report TR90-130, Dept. of Computer Science, Rice Univ., Houston, TX (1990).

Sel90a.

Selke, R.P., "Transforming Program Dependence Graphs," Technical Report TR90-131, Dept. of Computer Science, Rice Univ., Houston, TX (August 1990).

Seo88.

Seo, J. and Simmons, R., "Syntactic Graphs: A Representation for the Union of All Ambiguous Parse Trees," Tech. Rep. AI87-64, Artificial Intelligence Laboratory, University of Texas at Austin, Austin, TX (October 1988).

Sha81.

Sharir, M. and Pnueli, A., "Two Approaches to Interprocedural Data Flow Analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

Shi88.

Shivers, O., "Control Flow Analysis in Scheme," *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. 164-174 (July 1988).

Shi90.

Shivers, O., "Data-Flow Analysis and Type Recovery in Scheme," Tech. Report CMU-CS-90-115, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA (March 1990).

Sin72.

Sintzoff, M., "Calculating Properties of Programs by Valuations on Specific Models," pp. 203-207 in *Proceedings of an ACM Conference on Proving Assertions about Programs, (*Las Cruces, NM, January 6-7, 1972*), ACM SIGPLAN Notices*, (January 1972).

Son87.

Sondergaard, H. and Sestoft, P., *Non-determinacy and its Semantics*, Institute of Datalogy, University of Copenhagen, Copenhagen, Denmark (January 1987).

Str88.

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).

Str90.

Stransky, J., "A Lattice for Abstract Interpretation of Dynamic (Lisp-Like) Structures," LIX/RR/90/03, L.I.X., Ecole Polytechnique, Palaiseau, France (June 1990).

Tay80.

Taylor, R. N. and Osterweil, L. J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Transactions on Software Engineering* **SE-6**(3) pp. 265-277 (May 1980).

Ten74.

Tenenbaum, A., "Automatic Type Analysis in a Very High Level Language," Ph.D. dissertation, Computer Science Department, New York University, New York, NY (October 1974).

Tja70.

Tjaden, G.S. and Flynn, M.J., "Detection and Parallel Execution of Independent Interactions," *IEEE Trans. on Computers* C-19 pp. 889-895 (October 1970).

Tow76.

Towle, R., "Control and Data Dependence for Program Transformations," Ph.D. dissertation and Tech. Rep. R-76-788, Dept. of Computer Science, Univ. of Illinois, Urbana, IL (March 1976).

Wei80.

Weihl, W.E., "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables. ," pp. 83-94 in *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages*, (Las Vegas, NV, January 28-30, 1980), ACM, New York, NY (1980).

Wei84.

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

Wol91.

Wolf, M.E. and Lam, M.S., "A Data Locality Optimizing Algorithm," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto, Ontario, June 26-28, 1991), *ACM SIGPLAN Notices* 26(6) pp. 30-44 (June 1991).

Wol78.

Wolfe, M.J., "Techniques for Improving the Inherent Parallelism in Programs," Tech. Rep. R-78-929, Dept. of Computer Science, University of Illinois, Urbana, IL (July 1978).

Wol82.

Wolfe, M.J., "Optimizing Supercompilers for Supercomputers," Ph.D. dissertation and Tech. Rep. R-82-1105, Dept. of Computer Science, University of Illinois, Urbana, IL (October 1982).

Yan89.

Yang, W., Horwitz, S., and Reps, T., "Detecting program components with equivalent behaviors," TR-840, Computer Sciences Department, University of Wisconsin, Madison, WI (April 1989).

Yan90.

Yang, W., "A New Algorithm for Semantics-based Program Integration," Ph.D. dissertation and TR-962, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1990).

Index of Authors

Index of Terms and Definitions

List of Figures