ON THE ADEQUACY OF DEPENDENCE-BASED REPRESENTATIONS FOR PROGRAMS WITH HEAPS

by

Phil Pfeiffer & Rebecca Parsons Selke

Computer Sciences Technical Report #992

January 1991

On the Adequacy of Dependence-Based Representations for Programs with Heaps

Phil Pfeiffer
University of Wisconsin–Madison
pfeiffer@cs.wisc.edu

Rebecca Parsons Selke
Rice University
selke@rice.edu

Abstract. Program dependence graphs (*pdgs*) are popular tools for reasoning about a program's semantics. This report proves two fundamental theorems about the representational soundness of *pdgs* for languages with heap-allocated storage and reference variables. The first, the *Pointer-Language Equivalence Theorem*, asserts that *pdgs* adequately represent a program's meaning. The second, the *Pointer-Language Slicing Theorem*, asserts that *pdgs* adequately represent a program's threads of computation. These theorems are demonstrated with two new lemmas about the semantics of *pdgs* for languages that lack pointer variables. These lemmas, the *Dynamic Equivalence* and *Dynamic Slicing Theorems*, state that an edge can safely be removed from a program's *pdg* if this edge represents a static dependence that does not arise at run-time.

The proof of the Pointer-Language Equivalence Theorem assumes that computations have as much memory as they need to complete; *i.e.*, that reordering a computation's evaluation will not cause that computation to overflow its heap. It is argued that this assumption exposes a shortcoming of using dependences to reason about heap operations: standard notions of data dependence do not account for how programs consume storage.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, and Xerox.

1. INTRODUCTION

Program dependence graphs (*pdgs*) are popular tools for manipulating imperative programs. A *pdg* is a directed labeled graph that depicts a program's *dependences*—possible constraints on that program's evaluation. *Pdgs* have been used in optimizing and parallelizing compilers to restructure programs [Kuc81, All83, Fer87, Lar89], and in debuggers, program-integration tools, and semantic-differencing utilities to identify a program's threads of computation [Ott84, Cho89, Hor89, Hor90]. *Pdgs* are useful because they simplify program analysis; dependences abstract away from a program's statement-list operator, and expose its underlying threads of computation.

Early research on *pdgs* was limited to languages such as FORTRAN that support scalar variables and arrays, but lack heap-allocated storage. Recently, *pdgs* have been used to represent programs in *pointer languages*—languages such as Pascal that support heap-allocated storage, reference variables, and a destructive assignment statement. Several authors have proposed techniques for computing dependences in the presence of heaps [Hor89a, Lar89, Bod90, Gua90]. One of these authors [Lar89] also shows how dependences can be used to parallelize Lisp-like programs.

An important limitation of the work on pointer-language *pdgs* is the lack of theoretical justification for using *pdgs* to reason about programs with pointers. This report takes a first step towards providing such justification by proving two fundamental theorems about pointer-language *pdgs*. The first, the *Pointer-Language Equivalence Theorem*, states that two pointer-language programs that have isomorphic *pdgs* represent equivalent programs. The second, the *Pointer-Language Slicing Theorem*, states that a program's *pdg* adequately characterizes its threads of computation.

The Pointer-Language Equivalence and Slicing Theorems are proved by reducing assertions about the semantics of pointer-language *pdgs* to assertions about the semantics of *pdgs* for pointer-free languages. Example pointer-language programs are first reduced to programs in a simpler language (called $\mathcal{S}$) that supports neither references nor allocation. The reduction ensures that a reduced computation simulates the evaluation of an unreduced computation, up to the exhaustion of the simulated freelist. The reduction also ensures that programs with isomorphic *pdgs* are reduced to pointer-free programs with isomorphic *pdgs*. The problem of proving the Pointer-Language Equivalence and Slicing Theorems is therefore reduced to the task of proving two lemmas about the semantics of *pdgs* for $\mathcal{S}$-like languages:

* The first lemma, the *Dynamic Equivalence Theorem*, states that two programs in $\mathcal{S}$ that have isomorphic *dynamic pdgs* represent equivalent programs.

* The second lemma, the *Dynamic Slicing Theorem*, states that a *dynamic pdg* adequately represents a program's threads of computation.

These lemmas generalize existing adequacy theorems for $\mathcal{S}$-like languages by relaxing the requirements on a *pdg*'s set of edges. Earlier versions of the Equivalence and Slicing Theorems (*e.g.*, [Rep89]) assume that program P's *pdg* contains an edge $p \longrightarrow q$ if paths between p and q in P's control-flow graph satisfy certain criteria—even if these paths are never evaluated. The Dynamic Equivalence and Slicing Theorems, on the other hand, allow $p \longrightarrow q$ to be removed from P's *pdg* if none of P's possible evaluations exhibits $p \longrightarrow q$.

The proof of the Equivalence Theorem exposes an interesting shortcoming of using dependences to reason about programs with pointers. The proof assumes that a terminating computation can always allocate as much memory as it needs to complete. This assumption implies that any dependence-based reordering

of a computation's prescribed evaluation strategy is safe, regardless of whether that reordering increases a computation's peak memory requirements. The assumption that an evaluation strategy's memory requirements can safely be ignored may of course prove false for applications that use most of a machine's available memory. The assumption is needed, however, since standard notions of (data) dependence do not account for how operations consume storage.

The rest of the report contains four sections. Section 2 defines an example language whose features typify those exhibited by pointer languages. Section 3 defines a *pdg* for this language. Section 4 proves that this *pdg* provides an adequate representation of a program's semantics. Section 5 describes related research, and discusses the limitations of the theorems presented in this paper.

2. A LANGUAGE WITH HEAP-ALLOCATED STORAGE

This report argues that a dependence-graph representation of a program that uses pointers adequately characterizes that program's meaning and threads of computation. These claims are proved for the language $\mathcal{H}$, an example imperative language that exhibits the following features:

- Memory is represented as a map from locations to *structures*—cons cells and environments.
- Environments are maps from identifiers to *values* (*i.e.*, locations and atoms). Every store contains exactly one environment.
- Cons cells are maps from *selectors*—elements of { hd, tl }—to values.
- Programs are finite syntactic objects that map *finite* stores to stores. A store is finite if it contains finitely many structures that are accessible from (*i.e.*, can be reached from) that store's environment.
- Cons cells are allocated by expressions of the form *exp* :: *exp*.
- Storage is accessed by expressions such as x.hd and x.tl.
- Assignment statements alter the fields of environments and cons cells. For example, the statement x.hd := 0 overwrites the current contents of x.hd with the value 0.

The following grammar defines $\mathcal{H}$'s concrete syntax:

Program	→ *Stmt_list*
Stmt_list	→ *Stmt* {; *Stmt*}*
Stmt	→ **while** *Cond* **do** *Stmt_list* **od** | **if** *Cond* **then** *Stmt_list* **else** *Stmt_list* **fi** | *IdExp* := *Exp*
Cond	→ **isAtom** *simpleExp* | **isNil** *simpleExp* | **not** *Cond*
Exp	→ *SimpleExp* | *Exp* :: *Exp*
SimpleExp	→ ATOM | *IdExp*
IdExp	→ IDENT{.*Sel*}*
Sel	→ hd | tl

ATOM is a set of primitive objects—*e.g*, integers. IDENT is a set of lower-case alphanumeric identifier names. Members of IDENT are called *identifiers*. Members of *IdExp* are called *identifier expressions*.

Figure 1 gives an operational semantics for $\mathcal{H}$. Using an operational semantics to define $\mathcal{H}$ simplifies the task of reasoning about how programs evaluate.

The expression $P:\sigma$ is used to denote the sequence of states that results from evaluating the program P with respect to the store σ. The expression $P(\sigma)$ is used as a synonym for $\mathbf{M}_{\mathcal{H}}(P, \sigma)$, the *meaning* of the computation $P:\sigma$. $P(\sigma)$ is the error state $\perp$ if either $P:\sigma$ is not finite or $P:\sigma$ *terminates abnormally*—*i.e.*, if a statement either reads an undefined identifier, or applies a selector expression to an atom.

Three assumptions are made about the semantics of *alloc* ():

- *alloc* ( ) always returns an unused cons cell.
- *alloc* ( ) uses an auxiliary data structure known as the *freelist* to obtain unused cons cells. Each call to *alloc* ( ) removes the first cell from the freelist, and returns a reference to that cell.
- The freelist is unbounded: programs never fail for want of storage.

Although all three assumptions are typical of languages that support heap-allocated storage, the assumption that an implementation can support arbitrarily large stores is false in practice. How the difference between the idealized freelist and actual implementations of *alloc* affects the validity of these results is discussed in the final section of this paper.

$State = Point \times Store \quad Store = Loc \to Env + Cons \quad Env = Ident \to Val_\perp \quad Cons = Sel \to Val \quad Val = Loc + Atom$

$\mathbf{M}_{\mathcal{H}} : Prog \times Store \to Store_\perp$
$\mathbf{M}_{\mathcal{H}}(prog, \sigma) =$
let *freelist* be an unbounded list of locations that are not in $Domain(\sigma)$ **in**
let *firstPoint* be *prog*'s initial program point **in**
let $evalPgm = fix\ \lambda f.\ (\lambda((p, \sigma')) .\ p \stackrel{?}{=} \mathbf{final} \to \sigma' \,[]\, f(next(prog, (p, \sigma'))))$ **in** $evalPgm((firstPoint, \sigma))$
end*

$next : Prog \times State \to State_\perp$
$next(prog, (p, \sigma)) =$
case p **in**
$If(cexp), While(cexp)$: $(nextPoint(prog, p, cond(\sigma, cexp)), \sigma)$
$Assign(lexp, rexp)$: $(nextPoint(prog, p), assign(\sigma, lexp, rexp))$
esac

$cond : Store \times Cond \to Bool_\perp$
$cond(\sigma, cexp) =$
case $cexp$ **in**
isNil(exp): $(idexp(\sigma, exp) \stackrel{?}{=} \mathbf{nil})$
isAtom(exp): $(idexp(\sigma, exp) \in Atom)$
not(exp): $\neg\ cond(\sigma, exp)$
esac

$rvalue : Store \times Exp \to (Store \times Val)_\perp$
$rvalue(\sigma, rvexp) =$
case $rvexp$ **in**
Atom: $(\sigma, rvexp)$
Idexp: $(\sigma, idexp(\sigma, rvexp))$
$lexp :: rexp$: $cons(\sigma, lexp, rexp)$
esac

$assign : Store \times Exp \times Exp \to Store_\perp$
$assign(\sigma, lexp, rexp) =$
let $lv = idexp(\sigma, front(lexp))$ **in**
$lv \in Atom \to \perp\ []$
let $(\sigma', rv) = rvalue(\sigma, rexp)$ **in**
$\sigma'[\ (\sigma(lv))[rv\ /\ last(lexp)]\ /\ lv]$
end*

$cons : Store \times Exp \times Exp \to (Store \times Val)_\perp$
$cons(\sigma, lexp, rexp) =$
let $(\sigma', headv) = rvalue(\sigma, lexp)$ **in**
let $(\sigma'', tailv) = rvalue(\sigma', rexp)$ **in**
let $loc = alloc()$ **in**
$(\sigma''[\,[\mathrm{hd} \mapsto headv, \mathrm{tl} \mapsto tailv]\ /\ loc], loc)$
end*

$idexp : Store \to (Idexp \cup \{\varepsilon\}) \to Val_\perp$
$idexp(\sigma, idexpr) =$ **let** *env* be the location of the unique environment in σ **in** $selexp(\sigma, env, idexpr)$ **end**

$selexp : Store \times Val \times (Ident + Sel)^* \to Val_\perp$
$selexp(\sigma, val, selexpr) = selexpr \stackrel{?}{=} \varepsilon \to val\ []\ (val \in Atom \to \perp\ []\ selexp(\sigma, (\sigma(val))(first(selexpr)), tail(selexpr)))$

$alloc : () \to Loc = \lambda()\ .\ loc := first(freelist);\ freelist := tail(freelist);\ \mathbf{return}(loc)$

fix is the least fixpoint functional. $nextPoint(prog, p, cond)$ and $nextPoint(prog, p)$ denote point p's control-flow successors. ε is the empty sequence. $first(seq)$ and $last(seq)$ denote the first and last elements of sequence *seq*. $tail(seq)$ denotes all but the first, and $front(seq)$ all but the last, element of *seq*.

Figure 1. Language $\mathcal{H}$'s meaning function, $\mathbf{M}_{\mathcal{H}}$. Every function is strict in each argument.

3. PROGRAM DEPENDENCE GRAPHS

A *pdg* is a graph that depicts interactions between a program's component statements. *Pdgs* consist of nodes that represent a program's syntactic points of control, linked by edges $p \longrightarrow q$ that represent how one statement p might exchange information with, or control the evaluation of, a second statement q. The exact definition of a *pdg*—how its nodes and edges are labeled, and the notion of dependence it portrays—depends on the graph's intended use.

This report uses two kinds of *pdgs*. The first, the *dynamic heap pdg* (*hpdg*), represents programs in $\mathcal{H}$. The second, the *dynamic scalar pdg* (*spdg*), is used to prove assertions about *hpdgs*. Both *spdgs* and *hpdgs* are derived from a third *pdg*, referred to here as a *static pdg*. Static *pdgs* were developed for imperative languages with conditionals, loops, and scalar variables A static *pdg* for program P, G_P, contains one entry vertex, which represents P's initial point of control; one initial-definition and one final-use vertex for every variable in P; and one *if*, *while*, and *assignment* vertex for every *if* predicate, *while* predicate, and assignment in P, respectively. G_P also contains one edge for each of P's static *control*, *loop-carried flow*, *loop-independent flow*, *loop-carried def-order*, and *loop-independent def-order* dependences. The definitions of these dependences and the labeling requirements for the edges that represent them are summarized in Figure 2.

Let $level(p)$ be the number of *while* and *if* statements enclosing p. Program P has a ***control dependence*** $p \longrightarrow_c q$ iff either

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

P's static *pdg* contains one edge for each of P's control dependences. Edges that correspond to cases 1-3 are labeled **true**; edges that correspond to case 4 are labeled **false**.

P has a ***flow (output) dependence*** $p \longrightarrow_f q$ ($p \longrightarrow_o q$) iff there exists a path in P's control-flow graph from p to q such that p defines a variable x; x is not redefined between p and q; and q accesses (redefines) x. Dependence $p \longrightarrow q$ is ***carried*** by loop L iff L contains p and q, a path π gives rise to $p \longrightarrow q$, and π contains L's entry point. Dependence $p \longrightarrow q$ is ***loop-independent*** iff there exists a path π such that π gives rise to $p \longrightarrow q$ and, for all L that contain both p and q, π does not contain L's entry point.
(*N.B.*: A loop contains its own entry point—*i.e.*, its predicate vertex.)

P's *pdg* contains one edge $p \longrightarrow_{f(L)} q$ for every loop L that carries $p \longrightarrow_f q$. P's *pdg* also contains one edge $p \longrightarrow_{f(li)} q$ if $p \longrightarrow_f q$ is loop-independent.

P has a ***transitive*** output dependence $p_0 \longrightarrow_o^+ p_n$ iff there exist $p_2 \cdots p_{n-1}$ such that $p_0 \longrightarrow_o p_1 \cdots \longrightarrow_o p_n$. $p_0 \longrightarrow_o^+ p_n$ is carried by L if any of the $p_i \longrightarrow_o p_{i+1}$ are carried by L.

P has a ***def-order dependence*** $p \longrightarrow_{do(r)} q$ iff P has two dependences $p \longrightarrow_f r$ and $q \longrightarrow_f r$ through a variable x, and p precedes q in P's abstract syntax tree. P's *pdg* contains one edge $p \longrightarrow_{do(r,L)} q$ for every loop L that carries $p \longrightarrow_o^+ q$. P's *pdg* also contains one edge $p \longrightarrow_{do(r,li)} q$ if $p \longrightarrow_o^+ q$ is loop-independent.

Figure 2. Edges that make up a static *pdg*.

The only difference between a static *pdg* and an *spdg* is that an *spdg* portrays a *dynamic* notion of data dependence (*cf.* Figure 3). A static *pdg* for program P *must* contain the edge $p \longrightarrow q$ when P's control-flow graph contains paths from p to q that satisfy certain criteria—even if these paths are never evaluated. An *spdg* for P, on the other hand, may omit $p \longrightarrow q$ if none of P's *evaluations* exhibit $p \longrightarrow q$. An *spdg* for the program "[1] **if** *pred* **then** [2] $x := 1$ **fi**; [3] $y := x$", for example, may omit a flow edge from [2] to [3] when *pred* is uniformly false. Similarly, an *spdg* for a program P may omit $p \longrightarrow_{do\,(r)} q$ when none of P's evaluations exhibit both $p \longrightarrow_f r$ and $q \longrightarrow_f r$.

Hpdgs differ from *spdgs* in two ways. *Hpdgs* contain *one* initial definition vertex and *one* final use vertex, corresponding to every accessible structure in a program's initial and final stores, respectively. *Hpdgs* also portray a slightly different notion of data dependence—one that describes accesses of structures and fields (*cf.* Figure 4). To simplify the presentation—more specifically, the reduction described in Section 4—edges that arise from accesses of structures are omitted from *hpdgs*. Such edges can be omitted because they are accompanied by corresponding, transitive edges that arise from accesses of fields. This point may be illustrated by examining the dependences exhibited by an example program; for instance, "[1] $a :=$ **nil** :: **nil**; [2] $b := a$; [3] **if** *isAtom*(b) $\cdots$". This program exhibits [1] $\longrightarrow_f$ [3], since statement [3] checks the type of a cons cell that statement [1] creates. The program, however, also exhibits a transitive dependence [1] $\longrightarrow_f$ [2] $\longrightarrow_f$ [3] that arises from the accesses of a and b.

The definition of an *spdg* implies that programs can have multiple *spdgs*. Let P be a program that can be represented by a *pdg*. Let *static* (P) and *dynamic* (P) be program P's static and dynamic dependences, respectively. Let $S \subseteq$ *static* (P) be any *def-order-consistent* superset of *dynamic* (P); *i.e.*, any superset of *dynamic* (P) such that, for all p, q, r, and L, $p \longrightarrow_{do\,(r)} q \in S$ iff $p \longrightarrow_f r \in S \wedge q \longrightarrow_f r \in S$. Then the *spdg* that depicts only those data dependences in S is a valid *spdg* for program P. Similarly, let P_h be a program that manipulates pointers. Let *static* (P_h) be the set of dependences that P_h exhibits with respect to worst-

---

P exhibits a ***flow (output) dependence*** $p \longrightarrow_f q$ ($p \longrightarrow_o q$) iff there exists a store σ such that $P{:}\sigma$ generates a sequence of states $(p, \sigma_1) \cdots (q, \sigma_n)$; p defines a variable x in σ_1; x is not redefined between (p, σ_1) and (q, σ_n); and q accesses (redefines) x in σ_n. Dependence $p \longrightarrow q$ is ***carried*** by loop L iff L contains p and q, and there exists a store σ and a sequence of states $\pi = (p, \sigma_1) \cdots (q, \sigma_n)$ such that $P{:}\sigma$ generates π; π gives rise to $p \longrightarrow q$; and π contains a state at L's entry point. Dependence $p \longrightarrow q$ is ***loop-independent*** if there exists a store σ and a sequence of states $\pi = (p, \sigma_1) \cdots (q, \sigma_n)$ such that π gives rise to $p \longrightarrow q$ and, for all L that contain both p and q, π does not contain a state at L's entry point.
(*N.B.*: π contains a state at point p iff $\pi = (p_1, \sigma_1) \cdots (p_n, \sigma_n)$, and $p = p_i$ for some i between 1 and n inclusive.)

P's *pdg* contains one edge $p \longrightarrow_{f(L)} q$ for every loop L that carries $p \longrightarrow_f q$. P's *pdg* also contains one edge $p \longrightarrow_{f(li)} q$ if $p \longrightarrow_f q$ is loop-independent.

P exhibits a ***def-order dependence*** $p \longrightarrow_{do\,(r)} q$ iff P exhibits two dependences $p \longrightarrow_f r$ and $q \longrightarrow_f r$ through a variable x, and p precedes q in P's abstract syntax tree. P's *pdg* contains one edge $p \longrightarrow_{do\,(r,L)} q$ for every loop L that carries $p \longrightarrow_o^+ q$. P's *pdg* also contains one edge $p \longrightarrow_{do\,(r,li)} q$ if $p \longrightarrow_o^+ q$ is loop-independent.
(*N.B.*: The definition of transitive output dependence is similar to the definition given in Figure 2.)

Figure 3. Data-dependence edges that must be included in an *spdg*.

case aliasing assumptions about P_h. Let $S_h \subseteq static(P_h)$ be any def-order-consistent superset of $dynamic(P_h)$. Then the *hpdg* that depicts only those data dependences in S_h is valid for program P_h.

The most accurate representation of a program P's dependences is given by that dynamic pdg that depicts only those data dependences in $dynamic(P)$. A program's dynamic dependences, unfortunately, are not always computable. This assertion follows from the observation that it is not always possible to determine which of a program's statements will evaluate [Man74]. (*N.B.:* Dependence-computation algorithms for pointer languages typically compute a consistent superset of $dynamic(P)$; see, for example, [Hor89a].)

4. THE ADEQUACY OF HPDGS

This section proves two theorems about *hpdgs*. The first, the *Pointer-Language Equivalence Theorem*, shows that an *hpdg* provides an adequate characterization of a program's meaning. The second, the *Pointer-Language Slicing Theorem*, shows that an *hpdg* provides an adequate characterization of its threads of computation. Both theorems are proved by reducing programs that contain identifier and *cons* ("::") expressions to programs that lack these expressions. The reduced programs' *spdgs* are then compared, and conclusions drawn about the semantics of the original programs.

Intuitively, the reduction used to prove the Equivalence and Slicing Theorems is an algorithm for implementing language $\mathcal{H}$ in a second, reference-free language. The concrete syntax of the reduction's target language, $\mathcal{S}$, is given below:

Program	→ *Stmt_list*
Stmt_list	→ *Stmt* {; *Stmt*}*
Stmt	→ **while** *Cond* **do** *Stmt_list* **od** | **if** *Cond* **then** *Stmt_list* **else** *Stmt_list* **fi** | VAR := *Exp*
	→ **case** VAR **in** {REF : *Stmt_List*}* **esac**
Cond	→ VAR ∈ ATOM | VAR $\stackrel{?}{=}$ *Exp* | **not** *Cond*
Exp	→ *SimpleExp* | PRIMFN (*SimpleExp*)
SimpleExp	→ VAR | ATOM | REF
REF	→ ***1** | ***2** | ···

VAR is a set of alphanumeric variable names. Variables that are special to the reduction are given upper-

---

P exhibits a ***flow (output) dependence*** $p \longrightarrow_f q$ ($p \longrightarrow_o q$) iff there exists a store σ such that $P{:}\sigma$ generates a sequence of states $(p, \sigma_1) \cdots (q, \sigma_n)$; p defines a structure or field x in σ_1; x is not redefined between (p, σ_1) and (q, σ_n); and q accesses (redefines) x in σ_n.

P's *pdg* contains one edge $p \longrightarrow_{f(L)} q$ for every loop L that carries $p \longrightarrow_f q$. P's *pdg* also contains one edge $p \longrightarrow_{f(li)} q$ if $p \longrightarrow_f q$ is loop-independent.

P exhibits a ***def-order dependence*** $p \longrightarrow_{do(r)} q$ iff P exhibits two dependences $p \longrightarrow_f r$ and $q \longrightarrow_f r$ through a field x, and p precedes q in P's abstract syntax tree. P's *pdg* contains one edge $p \longrightarrow_{do(r,L)} q$ for every loop L that carries a $p \longrightarrow_o^+ q$ that arises through x. P's *pdg* also contains one edge $p \longrightarrow_{do(r,li)} q$ if a $p \longrightarrow_o^+ q$ that arises through x is loop-independent.

(*N.B.:* The definitions of loop-carried, loop-independent, and transitive dependence are similar to those given in Figures 2 and 3.)

Figure 4. Data-dependence edges that must be included in an *hpdg*.

case names; note that such names are not members of IDENT. PRIMFN is a collection of value-returning, side-effect-free, non-nested functions. The subscripts $\mathcal{H}$ and $\mathcal{S}$ are used when needed to distinguish between semantic objects in $\mathcal{H}$ and $\mathcal{S}$.

An operational semantics is assumed for $\mathcal{S}$, similar to one given in [Sel90]. Conditional expressions, *while* loops, assignment statements, and statement lists have their usual meaning. The *case* statement is equivalent to a nested if-then-else statement that causes a program to fail if none of its guards are matched. Language $\mathcal{S}$ has one non-standard feature: its meaning function maps a computation $\bar{P}:\bar{\sigma}$ to $\perp$ when $\bar{P}$ and $\bar{\sigma}$ fail to satisfy certain consistency constraints. These constraints, which ensure that every computation in $\mathcal{S}$ corresponds to some computation in $\mathcal{H}$, are described below.

Two functions are used to reduce computations in $\mathcal{H}$ to computations in $\mathcal{S}$. The one, $reduceStore_k$, reduces a $\sigma \in Store_{\mathcal{H}}$ to a comparable $\bar{\sigma} \in Store_{\mathcal{S}}$. Store $\bar{\sigma}$ consists of three sets of variables. The first contains one variable x for every identifier x in σ's environment. The second contains $2n$ variables that correspond to the n accessible cons cells in σ's heap: every accessible cell is reduced to a unique pair of variables (L_jHD, L_jTL), where j is an arbitrary value between 1 and n. The third set, which contains $2k$ variables named $L_{n+1}HD \cdots L_{n+k}TL$, simulates σ's infinitely-long freelist.

Function *reduceStore* reduces atoms to atoms. If σ's environment, for example, maps the identifier x to the value "3", then variable x in $\bar{\sigma}$ contains "3". References to the jth cons cell in σ are reduced to the value ***j** $\in$ REF. If σ's environment, for example, maps x to the jth cons cell in σ, then variable x in $\bar{\sigma}$ contains ***j**.

Function *reduceStore*, in effect, maps a $\sigma \in Store_{\mathcal{H}}$ to an *equivalent* $\bar{\sigma} \in Store_{\mathcal{S}}$. Two stores σ and $\bar{\sigma}$ are equivalent, written $\sigma \approx_{\mathcal{HS}} \bar{\sigma}$, if for all *idexp* and *idexp'* in *IDEXP*:

- *idexp* denotes the atom *at* in σ iff *idexp* corresponds to a variable v in $\bar{\sigma}$ that contains *at*. [1]
- *idexp* denotes a cons cell in σ iff *idexp* corresponds to a variable in $\bar{\sigma}$ that contains a ***j** $\in$ REF.
- *idexp* and *idexp'* denote the same structure in σ iff *idexp* and *idexp'* correspond to variables in $\bar{\sigma}$ that contain the same ***j** $\in$ REF.

The two other equivalence relations given in this report are similar to $\approx_{\mathcal{HS}}$. The one, $\approx_{\mathcal{H}}$, identifies isomorphic members of $Store_{\mathcal{H}}$. The other, $\approx_{\mathcal{S}}$, identifies members of $Store_{\mathcal{S}}$ that have comparable interpretations. The definitions of $\approx_{\mathcal{H}}$, $\approx_{\mathcal{HS}}$, and $\approx_{\mathcal{S}}$ ensure that $\sigma \approx_{\mathcal{H}} \sigma'$ when there exist $\bar{\sigma}$ and $\bar{\sigma'}$ such that $\sigma \approx_{\mathcal{HS}} \bar{\sigma} \approx_{\mathcal{S}} \bar{\sigma'} \approx_{\mathcal{HS}} \sigma'$.

Function *reducePgm*, the other reduction function, maps a $P \in Program_{\mathcal{H}}$ to a comparable $\bar{P} \in Program_{\mathcal{S}}$. The expression $reducePgm_{n,k,ident}(P)$ denotes a $\bar{P}$ that comprises a prologue and a body. Program $\bar{P}$'s prologue is a sequence of assignment statements that initialize a special variable, *NFREE*, to n, and every $x \in ident$ to the special atom *undefined*. Program $\bar{P}$'s body is obtained by recursively reducing P according to the rules given below:

A statement list "$stmt_1$; $\cdots$; $stmt_s$" is reduced by reducing each $stmt_i$ individually, according to the following rules for reducing statements.

---

[1] *idexp* corresponds to v in $\bar{\sigma}$ if $v = idexp$, or if $v = L_jSEL$, *idexp* is of the form $x.sel_1 \cdots sel_{n-1}SEL$, and $x.sel_1 \cdots sel_{n-1}$ denotes a variable containing the value ***j**.

An assignment such as "$x := at$", where at is an atom, is reduced to itself.

An assignment such as "$x.hd := y.tl$" is reduced to a nested **case** statement that first determines which variable corresponds to $y.tl$, then assigns the contents of this variable to the variable corresponding to $x.hd$:

case y **in**
 ***1:** **case** x **in** ***1:** $L_1HD := L_1TL$; ***2:** $L_2HD := L_1TL$; $\cdots$ ***n+k:** $L_{n+k}HD := L_1TL$; **esac**
$\cdots$
 ***n+k:** **case** x **in** ***1:** $L_1HD := L_{n+k}TL$; ***2:** $L_2HD := L_{n+k}TL$; $\cdots$ ***n+k:** $L_{n+k}HD := L_{n+k}TL$; **esac**
esac

Note that this reduction proscribes the use of an infinite simulated address space, since n and k must be finite to ensure that reduced programs contain finitely many points.

An assignment such as "$x := lexp::rexp$" is reduced to the four-statement sequence "$NFREE := NFREE+1$; $x := makeRef\,(NFREE)$; $x.hd := lexp$; $x.tl := rexp$". Variable $NFREE$ simulates P's internal pointer to the head of a store's freelist. Function $makeRef\,(NFREE)$ returns ***NFREE** when $NFREE \leq n+k$, and *undefined* otherwise. The assignments "$x.hd := lexp$" and "$x.tl := rexp$" must then be reduced by applying the reductions given above.

A conditional statement such as "**if** $isAtom\,(lexp)$ **then** SL **fi**" is first reduced to the pair of statements "$TEMP := lexp$; **if** $TEMP \notin$ REF **then** SL **fi**". Statement list SL and the assignment to $TEMP$ must then be reduced by applying the reductions given in this section.

The proofs of the Pointer-Language Equivalence and Slicing Theorems impose two requirements on the reduction from $\mathcal{H}$ to $\mathcal{S}$. The first requirement is that (1) reduced computations must mimic unreduced computations up to heap overflow. Requirement (1) is a two-part requirement:

(1a) A computation $\bar{P}{:}\bar{\sigma}$ must denote $\perp$ when either $\bar{P}$ names an identifier that is not present in $\bar{\sigma}$, or the initial configuration of $\bar{P}{:}\bar{\sigma}$'s address space or freelist is invalid; for example, when any of the simulated cons cells in $\bar{\sigma}$'s simulated initial freelist can be accessed by an identifier expression. Requirement (1a) ensures that every reduced computation in $\mathcal{S}$ corresponds to a valid computation in $\mathcal{H}$.

(1b) If σ is a store and $\bar{\sigma} = reduceStore_k\,(\sigma)$ for a suitable k, then $P{:}\sigma$ and $\bar{P}{:}\bar{\sigma}$ must generate comparable sequences of states and equivalent final stores—unless $\bar{P}{:}\bar{\sigma}$ fails by overflowing its simulated heap.

Requirement (1a) can be satisfied by defining $\mathcal{S}$'s meaning function so that it maps invalid combinations of stores and programs to $\perp$. Requirement (1b) can be satisfied by using *reducePgm* and *reduceStore* to define a reduction from $\mathcal{H}$ to $\mathcal{S}$, as follows:

DEFINITION. Let $P \in Program_{\mathcal{H}}$ and $\sigma \in Store_{\mathcal{H}}$. Let n be the number of accessible cons cells in σ. Let *idset* be the set of all identifiers not defined in σ, but referenced in P. Let k be an integer, $\bar{\sigma} = reduceStore_k\,(\sigma)$, and $\bar{P} = reducePgm_{n,k,idset}\,(P)$. The expression $reduce_k\,(P{:}\sigma)$ denotes $\bar{P}{:}\bar{\sigma}$. The expression $reduce_k\,(P(\sigma))$ denotes $\bar{P}(\bar{\sigma})$. □

LEMMA 1. Let $P \in Program_{\mathcal{H}}$, $\sigma \in Store_{\mathcal{H}}$, and k be a non-negative integer. Then $reduce_k\,(P{:}\sigma)$ fails if $P{:}\sigma$ allocates more than k cons cells; otherwise, $P{:}\sigma$ and $reduce_k\,(P{:}\sigma)$ generate corresponding sequences of states, with $P(\sigma) \approx_{\mathcal{HS}} reduce_k\,(P(\sigma))$.

PROOF. Lemma 1 is proved by induction on the number of steps in $P{:}\sigma$. This induction shows that P and $\bar{P}$ generate equivalent sequences of states, so long as $\bar{P}{:}\bar{\sigma}$ does not exhaust its simulated freelist. □

Program P		Program $\overline{P}$		Program $\overline{P'}$	
		[0]	$NFREE := 0$	[0]	$NFREE_1 := 0; NFREE_2 := 0$
[1]	$a := 7 :: 7$	[1.1]	$NFREE := NFREE + 1;$	[1.1]	$NFREE_1 := NFREE_1 + 1$
		[1.2]	$a := makeRef\ (NFREE);$	[1.2]	$a := free_1\ (NFREE_1)$
		[1.3]	**case** a **in**	[1.3]	**case** a **in**
			***1**: $L_1HD := 7;\ L_1TL := 7$		***1**: $L_1HD := 7;\ L_1TL := 7$
			...		...
			***n+k**: $L_{n+k}HD := 7;\ L_{n+k}TL := 7$		***n+k**: $L_{n+k}HD := 7;\ L_{n+k}TL := 7$
			esac		**esac**
[2]	$b := 8 :: 8$	[2.1]	$NFREE := NFREE + 1;$	[2.1]	$NFREE_2 := NFREE_2 + 1$
		[2.2]	$b := makeRef\ (NFREE);$	[2.2]	$b := free_2\ (NFREE_2)$
		[2.3]	**case** b **in**	[2.3]	**case** b **in**
			***1**: $L_1HD := 8;\ L_1TL := 8$		***1**: $L_1HD := 8;\ L_1TL := 8$
			...		...
			***n+k**: $L_{n+k}HD := 8;\ L_{n+k}TL := 8$		***n+k**: $L_{n+k}HD := 8;\ L_{n+k}TL := 8$
			esac		**esac**
				$free_1 = \lambda x\,.\,x \stackrel{?}{=} 1 \rightarrow$ ***1** [] *undefined*	
				$free_2 = \lambda x\,.\,x \stackrel{?}{=} 1 \rightarrow$ ***2** [] *undefined*	

Figure 5. Two reductions of "::" that produce different dependences. Program $\overline{P}$ exhibits [1.1] $\longrightarrow_f$ [2.1] through *NFREE*. Program $\overline{P'}$, which has two different freelist variables, does not exhibit [1.1] $\longrightarrow_f$ [2.1].

The proofs of the Pointer-Language Equivalence and Slicing Theorems also require that (2) P and $\overline{P}$ exhibit *comparable* dependences. More formally, let $R(s)$ be the set of program points in $\overline{P}$ that correspond under this reduction to a point s in P. Let p and q be distinct points in $\overline{P}$ such that $p \not\rightarrow_c q$. Then, for all $\overline{p} \in R(p)$ and all $\overline{q} \in R(q)$, the reduction must ensure that $\overline{p} \not\rightarrow_c \overline{q}$. Similar requirements hold for flow and def-order dependences. Intuitively, requirement (2) allows programs with isomorphic *hpdgs* to be reduced to programs with isomorphic *spdgs*.

The reduction defined in Lemma 1, unfortunately, fails to satisfy requirement (2). The reduction's use of a global freelist to parcel out storage, which provides a reasonable model of how programs allocate cells, can introduce unwanted dependences into reduced programs. Figure 5 illustrates an example reduced program, $\overline{P}$, that exhibits a flow dependence ([1.1] $\longrightarrow_f$ [2.1]) that corresponds to none of the dependences in the original program, P. This new dependence, which arises from [2.1]'s read of *NFREE*, represents a needless constraint on the order in which a program removes cells from its (simulated) freelist.

A second reduction depicted in Figure 5 (*cf.* program $\overline{P'}$) eliminates freelist-related dependences by splitting P's freelist into a set of local freelists—one per program point. Lemma 2 shows that the alternative reduction yields (1) an equivalent computation to the one obtained from the standard reduction of *cons*, and (2) a program whose dependences are comparable to those of the original program.

DEFINITION. Let program P contain p points. Let $free_1 \cdots free_p$ be functions from *Nat*, the natural numbers, to $\{n+1, \cdots, n+k, undefined\}$. The expression $reducePgm'_{n,k,ident}(P, free_1, \cdots, free_p)$ denotes a program that differs from $reducePgm_{n,k,ident}(P)$ in the following two ways:

- The statement "$NFREE := 0$" in P's prologue is replaced by p statements that set the variables $NFREE_1 \cdots NFREE_p$ to 0.

- An assignment such as "$[q]\ x := lexp::rexp$" is reduced to "$NFREE_q := NFREE_q + 1$; $x := free_q\,(NFREE_q)$; $x.hd := lexp$; $x.tl := rexp$". □

DEFINITION. Let $free_1 \cdots free_p$ be functions from *Nat* to *Nat* $\cup$ { *undefined* }. The $free_j$ are *mutually independent* iff $free_d\,(f) = free_e\,(g)$ implies that either $free_d\,(f) = undefined$ or $d = e$ and $f = g$. □

LEMMA 2. Let $P \in Program_{\mathcal{H}}$, $\sigma \in Store_{\mathcal{H}}$, and k be a non-negative integer. Let $\bar{\sigma} = reduceStore_k\,(\sigma)$. Let $free_1 \cdots free_p$ be mutually independent functions from *Nat* to $\{n+1, \cdots, n+k, undefined\}$. Let $\overline{P} = reducePgm_{n,k,ident}\,(P)$ and $\overline{P'} = reducePgm'_{n,k,ident}\,(P, free_1, \cdots, free_p)$. Then $\overline{P}{:}\bar{\sigma}$ and $\overline{P'}{:}\bar{\sigma}$ generate equivalent sequences of stores, unless $\overline{P'}{:}\bar{\sigma}$ fails because of the evaluation of a variable containing the value *undefined* (*i.e.*, because of a call to a $free_j$). Furthermore, P and $\overline{P'}$ have comparable dependences.

PROOF. The assertion that $\overline{P}{:}\bar{\sigma}$ and $\overline{P'}{:}\bar{\sigma}$ generate equivalent sequences of stores (up to the possible failure of $\overline{P'}{:}\bar{\sigma}$) follows from the independence of the $free_i$ and the referential transparency of $\mathcal{H}$.

The assertion that P and $\overline{P'}$ have comparable dependences is proved by reasoning about the reduction. The claim that P and $\overline{P'}$ have comparable sets of control dependences follows directly from the definition of *reducePgm*. The claim that P and $\overline{P'}$ also have comparable data dependences is proved by using P's *hpdg* to reason about the dynamic dependences exhibited by $\overline{P'}$. Figure 6 illustrates how a program $\overline{P'}$ may have static data dependences that fail to correspond to any of P's dependences, due to the use of *case* statements to simulate the interpretation of identifier expressions. Static dependences in $\overline{P'}$ of the form $[1.i] \longrightarrow_f [3.i]$ fail to correspond to dependences in P (*i.e.*, to $[1] \longrightarrow_f [3]$) when a and b are always aliased. Static dependences in $\overline{P'}$ of the form $[2.i] \longrightarrow_f [3.i]$ fail to correspond to dependences in P (*i.e.*, to $[2] \longrightarrow_f [3]$) when a and b are never aliased. Finally, static dependences in $\overline{P'}$ of the form $[1.i] \longrightarrow_{do([3.i])} [2.i]$ fail to correspond to dependences in P (*i.e.*, to $[1] \longrightarrow_{do([3])} [2]$) unless P exhibits both $[1] \longrightarrow_f [3]$ and $[2] \longrightarrow_f [3]$.

It can be shown, however, that $\overline{P'}$ and P exhibit comparable sets of dynamic data dependences. Let H_P be an arbitrary *hpdg* for P. Let $depend(H_P)$ be the set of data dependences depicted in H_P. Let $static\,(\overline{P'})$ be the set of static data dependences exhibited by $\overline{P'}$. Construct $induced\,(\overline{P'}, H_P)$ from $static\,(\overline{P'})$ by removing $\bar{p} \longrightarrow \bar{q}$ from $static\,(\overline{P'})$ when there exist p and q in P such that $p \neq q$, $\bar{p} \in R\,(p)$, $\bar{q} \in R\,(q)$, and $p \longrightarrow q \notin depend(H_P)$. Lemma 2 may now proved by arguing that:

Program P	[1] $a.hd := 7$; [2] $b.hd := 8$; [3] $a := a.hd$
Program $\overline{P}$	**case** a **in** [1.1] ***1**: $L_1HD := 7$ $\cdots$ [1.n+k] ***n+k**: $L_{n+k}HD := 7$ **esac** **case** b **in** [2.1] ***1**: $L_1HD := 8$ $\cdots$ [2.n+k] ***n+k**: $L_{n+k}HD := 8$ **esac** **case** a **in** [3.1] ***1**: $a := L_1HD$ $\cdots$ [3.n+k] ***n+k**: $a := L_{n+k}HD$ **esac**

Figure 6. Illustration of how the reduction adds static dependences to a program. A static dependence of the form $[1.i] \longrightarrow_f [3.i]$ will fail to correspond to a dependence in P (*i.e.*, to $[1] \longrightarrow_f [3]$) when a and b are always aliased. Similar observations hold for the $[2.i] \longrightarrow_f [3.i]$ and the $[1.i] \longrightarrow_{do([3.i])} [2.i]$.

i. $induced(\overline{P'}, P)$ contains all $\overline{P'}$'s dynamic dependences, and

ii. $depend(H_P)$ and $induced(\overline{P'}, H_P)$ are comparable sets of dependences.

Proof of (i). Let $dynamic(\overline{P'})$ denote $\overline{P'}$'s set of dynamic dependences. Let $removed(\overline{P'}, H_P)$ denote $static(\overline{P'}) - induced(\overline{P'}, H_P)$. Since $dynamic(\overline{P'}) \subseteq static(\overline{P'})$, claim (i) can be proved by showing that (*) $dynamic(\overline{P'}) \cap removed(\overline{P'}, H_P) = \emptyset$. To see the correctness of claim (*), assume to the contrary that some $\overline{p} \longrightarrow \overline{q}$ in $dynamic(\overline{P'})$ was removed from $static(\overline{P'})$. By the definition of a dynamic data dependence, there exists some $\overline{\sigma'}$ such that $\overline{P'}:\overline{\sigma'}$ exhibits $\overline{p} \longrightarrow \overline{q}$. However, the definition of $\mathcal{S}$'s meaning function (*cf.* requirement (1a)) and Lemma 1 then ensure the existence of a σ such that $P:\sigma$ exhibits $p \longrightarrow q$. Then $p \longrightarrow q \in depend(P)$, and $\overline{p} \longrightarrow \overline{q}$ could not have been removed from $static(\overline{P'})$—a contradiction.

Proof of (ii). Claim (ii) is proved by showing that $p \longrightarrow q \notin depend(H_P)$ iff $\overline{p} \longrightarrow \overline{q} \notin induced(\overline{P'}, H_P)$ for all $\overline{p} \in R(p)$ and $\overline{q} \in R(q)$. The *only if* direction—$p \longrightarrow q \notin depend(P)$ implies $\overline{p} \longrightarrow \overline{q} \notin induced(\overline{P'}, H_P)$—is immediate from the definition of $induced(\overline{P'}, H_P)$. To show the *if* direction, assume that $p \longrightarrow q \in depend(H_P)$. By the definition of data dependence, P's control-flow graph must contain a path from p to q. By the definition of the reduction, $\overline{P'}$'s control-flow graph must also contain paths from the points in $R(p)$ to the points in $R(q)$. By the definition of the reduction, there must be a $\overline{p} \in R(p)$ and a $\overline{q} \in R(q)$ such that $\overline{P'}$ exhibits a static dependence from $\overline{p}$ to $\overline{q}$. Then, by the definition of $induced$, $\overline{p} \longrightarrow \overline{q} \in induced(\overline{P'}, H_P)$. □

The proof of Lemma 2 concludes the characterization of the reduction proper. The proofs of the Pointer-Language Equivalence and Slicing Theorems use two additional lemmas, the Dynamic Equivalence and Slicing Theorems, to obtain a characterization of a reduced program's semantics. These lemmas are used to show that a reduced program's *spdg* provides an adequate characterization of its meaning and threads of computation, respectively.

The Dynamic Equivalence and Slicing Theorems are proved by using the graph-rewriting semantics for static *pdgs* given in [Sel89] to show that a data-dependence edge that does not correspond to a dynamic dependence can safely be removed from a program's *pdg*. This semantics is a *pdg*-rewriting system that identifies *redexes*—nodes with no remaining incoming dependences—and updates the *pdg* to reflect the impact of the node. For example, the rewriting of an assignment node causes the propagation of the value of its expression to all other nodes with incoming flow edges from the redex. In this semantics, nodes are removed from the *pdg* when they are rewritten, and nodes that do not execute due to the outcome of a predicate node are removed without being rewritten. A *rewriting sequence* shows the order in which the redexes are rewritten. The result of a rewriting is the set of final use nodes containing identifiers and their final values.

LEMMA (*Dynamic Equivalence Theorem*). Suppose that P and Q are members of $Program_{\mathcal{S}}$ that have isomorphic *spdgs*. Let σ and σ' agree on the values of all variables named by P's initial-definition vertices. If P halts on σ then (1) Q halts on σ', (2) P and Q compute identical sequences of values at corresponding program points, and (3) P and Q compute stores that agree on the values of all store-access expressions named by P's final-use vertices.

PROOF. Since the *spdgs* for P and Q are isomorphic, their meanings under the rewriting semantics from [Sel89] are the same. If the rewriting semantics holds for *spdgs* as well as *pdgs*, the lemma is shown. The proof for the following claim appears below.

CLAIM. Let P be a program in $Program_S$, let G be its static *pdg* and let G' be an *spdg* for P. Then G rewrites to the set $\{ (x_1, v_1) \cdots (x_n, v_n) \}$ for all x_i appearing in the final-use vertices iff G' rewrites to the same set.

PROOF OF CLAIM. By the definition of *spdgs*, G and G' have the same node set and control edge set. The proof proceeds by induction on the length of the rewriting sequence. The induction hypothesis includes the following three claims:

1. The same values are computed for corresponding redexes in G and G'.
2. The same values flow over corresponding edges in G and G'.
3. The nodes removed in G and G' by the rewriting are the same.

The base case is trivial. Since G and G' only differ in the flow and def-order edge sets, there are three cases to consider for the induction step: the same redex is available for rewriting in both G and G', the node in G corresponding to the redex in G' has an incoming flow edge, and the node in G corresponding to the redex in G' has an incoming def-order edge.

The first case is trivial.

For the second case, let n and n' be the nodes in G and G' respectively, and let (p,n) be the edge in G. Since this edge is not in G' and by the induction hypothesis, this edge must *not* be exhibited by any rewriting of P. Since n' is a redex, all control edges for n must have been resolved so n will execute. Thus, either (i) p does not send a value to n, or (ii) the value that p sends to n will be overwritten by some other node p'. If (i) holds, then n becomes a redex that is, by part 2 of the induction hypothesis, not different from n'. Otherwise, if (ii) holds, the def-order consistency condition for *pdgs* (*cf.* Section 3) and part 3 of the induction hypothesis ensure the presence of a def-order edge (p,p') in G. The value sent to n' by p' must therefore overwrite any value sent by p. By part 2 of the induction hypothesis, when n does become a redex, its expression will evaluate to the same value as n'.

The arguments for the third case are similar to those of the second case, but there are more possibilities to consider in the differences in the graphs. These differences arise because there are more situations where a def-order edge can be removed from the spdg than there are situations for flow edges.

With this claim, the lemma follows directly from the *pdg* Equivalence Theorem given in [Sel89]. □

DEFINITION. A *slice* of a *pdg* G with respect to vertex set S, written G / S, is the subgraph of G induced by $V(G / S)$, the set of vertices v such that there exists a path from v to a vertex in S—less all edges $u \rightarrow_{do\ (w)} v$ such that $w \notin V(G / S)$. □

Intuitively, G / S is safe approximation to the set of all points in G that could contribute to the sequence of values computed at points in S.

LEMMA (*Dynamic Slicing Theorem*). Let Q be a slice of a $P \in Program_S$ with respect to a set of vertices. Let σ and σ' agree on the values of all variables named by Q's initial-definition vertices. If P halts on σ, then (1) Q halts on σ', (2) P and Q compute identical sequences of values at each program point of Q, and (3) P and Q agree on the values of all variables named in Q's final-definition vertices.

PROOF. Let G and H be the *spdgs* for the program and the slice respectively. Since P halts, the rewriting sequence for G is finite. A finite rewriting sequence for H can be constructed from this sequence by simply selecting the steps for nodes that exist in H . From the definition of a slice, this sequence results in a final value for Q and is a valid rewriting sequence. Thus, Q halts. Since the steps are identical for all nodes in H, and since the rewriting semantics shows that the effects of rewriting a node are confined to nodes with

edges from the rewritten node, the rest of the theorem follows directly. □

This concludes the presentation of the auxiliary lemmas used in the Pointer-Language Equivalence and Slicing Theorems. Proofs of these theorems now follow.

THEOREM (*Pointer-Language Equivalence Theorem*). Let P and Q be programs that have isomorphic hpdgs, H_P and H_Q. If P terminates successfully on σ, then (1) Q terminates successfully on σ, (2) P and Q compute equivalent sequences of values at corresponding program points, and (3) $P(\sigma) \approx_{\mathcal{H}} Q(\sigma)$.

PROOF. Let *ident* be the set of identifiers named in P but not defined in σ. Let n be the number of accessible cells in σ. Let k be an upper bound on the number of cells that $P{:}\sigma$ allocates; such a bound must exist because P terminates. Let $\bar{\sigma} = reduceStore_k(\sigma)$ and $\bar{P} = reducePgm_{n,k,ident}(P)$. By Lemma 1, $\bar{P}(\bar{\sigma})$ succeeds, with $P(\sigma) \approx_{\mathcal{HS}} \bar{P}(\bar{\sigma})$. Since $\bar{P}(\bar{\sigma})$ terminates, there exist mutually independent $free_1 \cdots free_p$ and a $\overline{P'} = reducePgm'_{n,k,ident}(P, free_1, \cdots, free_p)$ such that $\bar{P}(\bar{\sigma}) \approx_{\mathcal{S}} \overline{P'}(\bar{\sigma})$.

Let $\overline{Q'} = reducePgm'_{n,k,ident}(Q, free_{\pi^{-1}(1)}, \cdots, free_{\pi^{-1}(p)})$, where $\pi(j)$ denotes that program point in H_Q that corresponds under the isomorphism to point j in H_P. If programs $\overline{P'}$ and $\overline{Q'}$ have isomorphic *spdgs*, then it follows from the Dynamic Equivalence Theorem that $\overline{P'}$ and $\overline{Q'}$ are equivalent programs.

Let $spdg(\overline{P'})$ denote that *spdg* for program $\overline{P'}$ that depicts only those data dependences in $induced(\overline{P'}, H_P)$ (*cf.* the proof of Lemma 2). Similarly, let $spdg(\overline{Q'})$ denote that *spdg* for $\overline{Q'}$ that depicts only those dependences in $induced(\overline{Q'}, H_Q)$. A two-part argument can now be used to show that $spdg(\overline{Q'})$ and $spdg(\overline{P'})$ are isomorphic. It must first be shown that (1) $\bar{P}$ and $\bar{Q}$ have isomorphic sets of program points. This claim, however, follows immediately from the choice of the $free_{\pi(j)}$. It must then be shown that (2) $spdg(\overline{P'})$ and $spdg(\overline{Q'})$ have isomorphic edge sets. The proof of assertion (2) can be divided into two cases. If $\bar{p}$ and $\bar{q}$ are two points in $spdg(\overline{P'})$ that correspond to a single point in P, then the reduction ensures that edges between $\bar{p}$ and $\bar{q}$ in $spdg(\overline{P'})$ are isomorphic to edges between $\pi(\bar{p})$ and $\pi(\bar{q})$ in $spdg(\overline{Q'})$. Otherwise, $\bar{p}$ and $\bar{q}$ correspond to distinct points in P. Then Lemma 2 and the reduction ensure that edges between $\bar{p}$ and $\bar{q}$ are isomorphic to edges between $\pi(\bar{p})$ and $\pi(\bar{q})$. Graph $spdg(\overline{Q'})$ is therefore isomorphic to $spdg(\overline{P'})$.

Since $\overline{P'}$ and $\overline{Q'}$ have isomorphic *spdgs*, the Dynamic Equivalence Theorem now implies that $\overline{Q'}(\bar{\sigma})$ succeeds, with $\overline{P'}(\bar{\sigma}) = \overline{Q'}(\bar{\sigma})$. (Computation $\overline{Q'}{:}\bar{\sigma}$ cannot overflow, since $\overline{P'}(\bar{\sigma})$ and $\overline{Q'}(\bar{\sigma})$ compute the same final values for the $NFREE_i$.)

Let $\bar{Q} = reducePgm_{n,k,ident}(Q)$. By Lemma 2, the mutual independence of the $free_i$, and the observation that $\overline{Q'}(\bar{\sigma})$ succeeds, $\bar{Q}(\bar{\sigma})$ must also succeed, with $\overline{Q'}(\bar{\sigma}) \approx_{\mathcal{S}} \bar{Q}(\bar{\sigma})$. Since $\bar{Q}(\bar{\sigma})$ terminates successfully, Lemma 1 now implies that $\bar{Q}(\bar{\sigma}) \approx_{\mathcal{HS}} Q(\sigma)$.

To summarize the preceding argument, $P(\sigma) \approx_{\mathcal{HS}} \bar{P}(\bar{\sigma}) \approx_{\mathcal{S}} \overline{P'}(\bar{\sigma}) = \overline{Q'}(\bar{\sigma}) \approx_{\mathcal{S}} \bar{Q}(\bar{\sigma}) \approx_{\mathcal{HS}} Q(\sigma)$. The definitions of the various equivalence relations now imply that $P(\sigma) \approx_{\mathcal{H}} Q(\sigma)$. An extension of this argument shows that P and Q generate equivalent values at corresponding program points. □

COROLLARY. If $\sigma \approx_{\mathcal{H}} \sigma'$, then (1) Q halts on σ', (2) P and Q compute corresponding sequences of values at corresponding program points, and (3) $P(\sigma) \approx_{\mathcal{H}} Q(\sigma')$.

PROOF. Since language $\mathcal{H}$ is referentially transparent, $P(\sigma) \approx_{\mathcal{H}} P(\sigma')$. The corollary now follows from the main theorem and the transitivity of $\approx_{\mathcal{H}}$. □

THEOREM (*Pointer-Language Slicing Theorem*). Let Q be a slice of a program P with respect to a set of vertices. Let σ and σ' be stores that agree on the values of all store-access expressions named by Q's

initial-definition vertices. If P halts on σ, then (1) Q halts on σ', (2) P and Q compute equivalent values on each program point of Q, and (3) P and Q agree on the values of all store-access expressions named in Q's final-definition vertices.

PROOF. This claim follows from the Dynamic Slicing Theorem and the assertion that the reduction based on *reducePgm'* maps dependences in a program to comparable dependences in a reduced program. □

5. DISCUSSION

This report presents what we believe are the first proofs of the Equivalence and Slicing Theorems, relative to a language with heap allocation and pointer variables. These proofs can be extended to an expanded language that supports other structures and referentially-transparent operators and predicates. The limitations imposed by the lack of multiple initial-definition and final-use vertices can be overcome by adding input and output statements to $\mathcal{H}$, using the techniques outlined in [Sel89].

The idea that a program's semantics can be represented by its data dependences was proposed by Kuck *et. al.* in [Kuc72]. The various kinds of dependence graphs that have been proposed since [Kuc72]—such as *pdgs* [Ott84, Fer87, Sel89, Hor89], *system dependence graphs* (*sdgs*) [Hor90a], *program representation graphs* (*prgs*) [Hor90], and *program dependence webs* [Bal90]—represent different extensions of [Kuc72]. None of the these representations were intended for languages with heaps.

Horwitz, Reps, and Prins 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 strengthed 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].

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].

Other reports on the semantics of dependence-graph representations include [Sel90a], which gives a calculus for *pdgs*, [Ram89], which gives a semantics for *prgs*, and [Bin89], which proves an equivalence theorem for *sdgs*—a *pdg*-like representation for languages with procedures. 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 used here.

The Dynamic Equivalence and Slicing Theorems are believed to be the first soundness theorems for dependence-based program representations that use a dynamic notion of data dependence.

Horwitz *et. al.* were the first to give a provably safe algorithm for computing a program's dependences for an $\mathcal{H}$-like language [Hor89a]. Larus was the first to describe how dependences can be used to find parallelism in Lisp-like programs [Lar89]. Bodin and Guarna have also given dependence-computation algorithms for programs with heaps and pointers [Bod90, Gua90]. None of these reports, however, show that dependence-based representations for programs are sound.

The theorems demonstrated in this report do not guarantee the soundness of many common dependence-based program transformations, such as loop unrolling and parallel evaluation (*cf.* [Lar89]). Such concerns, though important, are beyond the scope of this paper. The report also fails to consider whether the classic kinds of dependences that are represented in *pdgs* adequately portray how "real" pro-

grams manipulate storage. The theorems presented in Section 4 make claims about how programs evaluate when their freelists contain arbitrarily many cons cells. It can be argued that this assumption is not a crucial limitation, so long as a program's heap is large enough to support any of its feasible evaluation orders: the freelist may be viewed as a list of virtual cons cells, and a garbage-collector as an oracle that maps virtual cells onto free locations. This use of an oracle, however, begs the question of whether a statement that allocates cells should be regarded as being dependent on statements that deallocate cells. A more serious problem with the failure to account for storage limitations is that a naive, but apparently safe, rearrangement of a program's default execution order may cause that program to overflow its heap.

6. ACKNOWLEDGEMENTS

Thanks are extended to Tom Reps, Tom Ball, David Binkley, and G. Ramalingam for providing valuable critiques of this report. Thanks are also extended to David Chase, whose observations about how transformations affect storage consumption inspired the observations about heap overflow [Cha88].

REFERENCES

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).

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).

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).

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).

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).

Cho89.
Choi, J., "Parallel Program Debugging with Flowback Analysis," Ph.D. dissertation and TR-871, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1989).

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).

Gua90.
Guarna, V.A. Jr., "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).

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).

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)(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).

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).

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).

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).

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).

Ram89.
Ramalingam, G. and Reps, T., "Semantics of program representation graphs," TR-900, Computer Sciences Department, University of Wisconsin, Madison, WI (December 1989).

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).

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).