Building a Bridge between Pointer Aliases and Program Dependences *

John L. Ross[1] and Mooly Sagiv[2]

[1] University of Chicago, e-mail: johnross@cs.uchicago.edu
[2] Tel-Aviv University, e-mail: sagiv@math.tau.ac.il

Abstract. In this paper we present a surprisingly simple reduction of the program dependence problem to the may-alias problem. While both problems are undecidable, providing a bridge between them has great practical importance. Program dependence information is used extensively in compiler optimizations, automatic program parallelizations, code scheduling in super-scalar machines, and in software engineering tools such as code slicers. When working with languages that support pointers and references, these systems are forced to make very conservative assumptions. This leads to many superfluous program dependences and limits compiler performance and the usability of software engineering tools. Fortunately, there are many algorithms for computing conservative approximations to the may-alias problem. The reduction has the important property of always computing conservative program dependences when used with a conservative may-alias algorithm. We believe that the simplicity of the reduction and the fact that it takes linear time may make it practical for realistic applications.

1 Introduction

It is well known that programs with pointers are hard to understand, debug, and optimize. In recent years many interesting algorithms that conservatively analyze programs with pointers have been published. Roughly speaking, these algorithms [19, 20, 25, 5, 16, 17, 23, 8, 6, 9, 14, 27, 13, 28] conservatively solve the may-alias problem, i.e., the algorithms are sometimes able to show that two pointer access paths never refer to the same memory location at a given program point.

However, may-alias information is insufficient for compiler optimizations, automatic code parallelizations, instruction scheduling for super-scalar machines, and software engineering tools such as code slicers. In these systems, information about the program dependences between *different* program points is required. Such dependences can be uniformly modeled by the program dependence graph (see [21, 26, 12]).

In this paper we propose a simple yet powerful approach for finding program dependences for programs with pointers:

* Partially supported by Binational Science Foundation grant No. 9600337

> Given a program $\mathcal{P}$, we generate a program $\mathcal{P}'$ (hereafter also referred to as the *instrumented* version of $\mathcal{P}$) which simulates $\mathcal{P}$. The program dependences of $\mathcal{P}$ can be computed by applying an arbitrary conservative may-alias algorithm to $\mathcal{P}'$.

We are reducing the program dependence problem, a problem of great practical importance, to the may-alias problem, a problem with many competing solutions. The reduction has the property that as long as the may-alias algorithm is conservative, the dependences computed are also conservative. Furthermore, there is no loss of precision beyond that introduced by the chosen may-alias algorithm. Since the reduction is quite efficient (linear in the program size), it should be possible to integrate our method into compilers, program slicers, and other software tools.

1.1 Main Results and Related Work

The major results in this paper are:

- The unification of the concepts of program dependences and may-aliases. While these concepts are seemingly different, we provide linear reductions between them. Thus may-aliases can be used to find program dependences and program dependences can be used to find may-aliases.
- A solution to the previously open question about the ability to use "store-less" (see [8–10]) may-alias algorithms such as [9, 27] to find dependences. One of the simplest store-less may alias algorithm is due to Gao and Hendren [14]. In [15], the algorithm was generalized to compute dependences by introducing new names. Our solution implies that there is no need to re-develop a new algorithm for every may-alias algorithm. Furthermore, we believe that our reduction is actually simpler to understand than the names introduced in [15] since we are proving program properties instead of modifying a particular approximation algorithm.
- Our limited experience with the reduction that indicates that store-less may-alias algorithms such as [9, 27] yield quite precise dependence information.
- The provision of a method to compare the time and precision of different may-alias algorithms by measuring the number of program dependences reported. This metric is far more interesting than just comparing the number of may-aliases as done in [23, 11, 31, 30, 29].

Our program instrumentation closely resembles the "instrumented semantics" of Horwitz, Pfeiffer, and Reps [18]. They propose to change the program semantics so that the interpreter will carry-around program statements. We instrument the program itself to locally record statement information. Thus, an arbitrary may-alias algorithm can be used on the instrumented program without modification. In contrast, Horwitz, Pfeiffer, and Reps proposed modifications to the specific store based may-alias algorithm of Jones and Muchnick [19] (which is imprecise and doubly exponential in space).

An additional benefit of our shift from semantics instrumentation into a program transformation is that it is easier to understand and to prove correct. For example, Horwitz, Pfeiffer, and Reps, need to show the equivalence between the original and the instrumented program semantics and the instrumentation properties. In contrast, we show that the instrumented program simulates the original program and the properties of the instrumentation.

Finally, program dependences can also be conservatively computed by combining side-effect analysis [4, 7, 22, 6] with reaching definitions [2] or by combining conflict analysis [25] with reaching definitions as done in [24]. However, these techniques are extremely imprecise when recursive data structures are manipulated. The main reason is that it is hard to distinguish between occurrences of the same heap allocated run-time location (see [6, Section 6.2] for an interesting discussion).

1.2 Outline of the rest of this paper

In Section 2.1, we describe the simple Lisp-like language that is used throughout this paper. The main features of this language are its dynamic memory, pointers, and destructive assignment. The use of a Lisp-like language, as opposed to C, simplifies the presentation by avoiding types and the need to handle some of the difficult aspects of C, such as pointer arithmetic and casting.

In Section 2.2, we recall the definition of flow dependences. In Section 2.3 the may-alias problem is defined.

In Section 3 we define the instrumentation. We show that the instrumented program simulates the execution of the original program. We also show that for every run-time location of the original program the instrumented program maintains the history of the statements that last wrote into that location. These two properties allow us to prove that may-aliases in the instrumented program precisely determine the flow dependences in the original program.

In Section 4, we discuss the program dependences computed by some may-alias algorithms on instrumented programs. Finally, Section 5, contains some concluding remarks.

2 Preliminaries

2.1 Programs

Our illustrative language (following [19, 5]) combines an Algol-like language for control flow and functions, Lisp-like memory access, and explicit destructive assignment statements. The atomic statements of this language are shown in Table 1. Memory access paths are represented by $\langle Access \rangle$. Valid expressions are represented by $\langle Exp \rangle$. We allow arbitrary control flow statements using conditions $\langle Cond \rangle$[1]. Additionally all statements are labeled.

[1] Arbitrary expressions and procedures can be allowed as long as it is not possible to observe actual run-time locations.

Figure 1 shows a program in our language that is used throughout this paper as a running example. This program reads atoms and builds them into a list by destructively updating the *cdr* of the tail of the list.

```
    program DestructiveAppend()
    begin
s1:   new( head )
s2:   read( head.car )
s3:   head.cdr := nil
s4:   tail := head
s5:   while( tail.car ≠ 'x' ) do
s6:       new( temp )
s7:       read( temp.car )
s8:       temp.cdr := nil
s9:       tail.cdr := temp
s10:      tail := tail.cdr
      od
s11:  write( head.car )
s12:  write( tail.car )
    end.
```

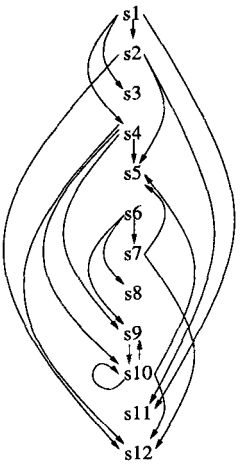

Fig. 1. A program that builds a list by destructively appending elements to *tail* and its flow dependences.

2.2 The Program Dependence Problem

Program dependences can be grouped into flow dependences (def-use), output dependences (def-def), and anti-dependences (use-def) [21, 12]. In this paper, we focus on flow dependences between program statements. The other dependences are easily handled with only minor modifications to our method.

Table 1. An illustrative language with dynamic memory and destructive updates.

$\langle St \rangle ::= s_i : \langle Access \rangle := \langle Exp \rangle$
$\langle St \rangle ::= s_i :$ **new**$(\langle Access \rangle)$
$\langle St \rangle ::= s_i :$ **read**$(\langle Access \rangle)$
$\langle St \rangle ::= s_i :$ **write**$(\langle Exp \rangle)$
$\langle Access \rangle ::= variable \mid \langle Access \rangle.\langle Sel \rangle$
$\langle Exp \rangle ::= \langle Access \rangle \mid atom \mid$ **nil**
$\langle Sel \rangle ::= car \mid cdr$
$\langle Cond \rangle ::= \langle Exp \rangle = \langle Exp \rangle$
$\langle Cond \rangle ::= \langle Exp \rangle \neq \langle Exp \rangle$

Our language allows programs to explicitly modify their store through pointers. Because of this we must phrase the definition of flow dependence in terms of memory *locations* (cons-cells) and not variable names. We shall borrow the following definition for flow dependence:

Definition 1 ([18]). *Program point q has a flow dependence on program point p if p writes into memory location loc that q reads, and there is no intervening write into loc along an execution path by which q is reached from p.*

Figure 1 also shows the flow dependences for the running example program. Notice that s_{11} is flow dependent on only s_1 and s_2, while s_{12} is flow dependent on s_2, s_4, s_7, and s_{10}. This information could be used by slicing tools to find that the loop need not be executed to print *head.car* in s_{11}, or by an instruction scheduler to reschedule s_{11} for anytime after s_2. Also, s_3 and s_8 have no statements dependent on them, making them candidates for elimination. Thus, even in this simple example, knowing the flow dependences would potentially allow several code transformations.

Since determining the exact flow dependences in an arbitrary program is undecidable, approximation algorithms must be used. A flow dependence approximation algorithm is *conservative* if it always finds a superset of the true flow dependences.

2.3 The May-Alias Problem

The may-alias problem is to determine whether two access-paths, at a given program point, could denote the same cons-cell.

Definition 2. *Two access-paths are may-aliases at program point p, if there exists an execution path to program point p where both denote the same cons-cell.*

In the running example program, *head.cdr.cdr* and *tail* are may-aliases at s_6 since before the third iteration these access paths denote the same cons-cell. However, *tail.cdr.cdr* is not a may-alias to *head* since they can never denote the same cons-cell.

Since the may-alias problem is undecidable, approximation algorithms must be used. A may-alias approximation algorithm is *conservative* if it always finds a superset of the true may-aliases.

3 The Instrumentation

In this section the instrumentation is defined. For notational simplicity, $\mathcal{P}$ stands for an arbitrary fixed program, and $\mathcal{P}'$ stands for its instrumented version.

3.1 The Main Idea

The program $\mathcal{P}'$ simulates all the execution sequences of $\mathcal{P}$. Additionally, the "observable" properties of $\mathcal{P}$ are preserved.

Most importantly, $\mathcal{P}'$ records for every variable v, the statement from $\mathcal{P}$ that last wrote into v. This "instrumentation information" is recorded in $v.car$ (while storing the original content of v in $v.cdr$). This "totally static" instrumentation[2] allows program dependences to be recovered by may-alias queries on $\mathcal{P}'$.

More specifically, for every statement in $\mathcal{P}$ there is an associated cons-cell in $\mathcal{P}'$. We refer to these as *statement* cons-cells. Whenever a statement s_i assigns a value into a variable v, $\mathcal{P}'$ allocates a cons-cell that we refer to as an *instrumentation* cons-cell. The *car* of this instrumentation cons-cell always points to the statement cons-cell associated with s_i. Thus there is a flow dependence from a statement p to q: $x := y$ in $\mathcal{P}$ if and only if $y.car$ can point to the statement cons-cell associated with p in $\mathcal{P}'$. Finally, we refer to the *cdr* of the instrumentation cons-cell as the *data* cons-cell. The data cons-cell is inductively defined:

- If s_i is an assignment s_i: $v := A$ for an atom, A, then the data cell is A.
- If s_i is an assignment s_i: $v := v'$ then the data cell denotes $v'.cdr$.
- If the statement is s_i: $new(v)$, then the data cons-cell denotes a newly allocated cons-cell. Thus $\mathcal{P}'$ allocates two cons-cells for this statement.

In general, there is an inductive syntax directed definition of the data cells formally defined by the function *txe* defined in Table 3.

3.2 The Instrumentation of the Running Example

To make this discussion concrete, Figure 2 shows the beginning of the running example program and its instrumented version. Figure 3 shows the stores of both the programs just before the loop (on the input beginning with 'A'). The cons-cells in this figure are labeled for readability only.

The instrumented program begins by allocating one statement cons-cell for each statement in the original program. Then, for every statement in the original program, the instrumented statement block in the instrumented program records the last wrote-statement and the data. The variable *_rhs* is used as a temporary to store the right-hand side of an assignment to allow the same variable to be used on both sides of an assignment.

Let us now illustrate this for the statements s_1 through s_4 in Figure 3.

- In the original program, after s_1, *head* points a new uninitialized cons-cell, c_1. In the instrumented program, after the block of statements labeled by s_1, *head* points to an instrumentation cons-cell, i_1, *head.car* points to the statement cell for s_1, and *head.cdr* points to c_1'.

[2] In contrast to dynamic program slicing algorithms that record similar information using hash functions, e.g., [1].

- In the original program, after s_2, *head.car* points to the atom A. In the instrumented program, after the block of statements labeled by s_2, *head.cdr.car* points to an instrumentation cons-cell, i_2, *head.cdr.car.car* points to the statement cell for s_2, and *head.cdr.car.cdr* points to A.
- In the original program, after s_3, *head.cdr* points to nil. In the instrumented program, after the block of statements labeled by s_3, *head.cdr.cdr* points to an instrumentation cons-cell, i_3, *head.cdr.cdr.car* points to the statement cell for s_3, and *head.cdr.cdr.cdr* points to nil.
- In the original program, after s_4, *tail* points to the cons-cell c_1. In the instrumented program, after the block of statements labeled by s_4, *tail* points to an instrumentation cons-cell, i_4, *tail.car* points to the statement cell for s_4, and *tail.cdr* points to c_1'. Notice how the sharing of the r-values of *head* and *tail* is preserved by the transformation.

3.3 A Formal Definition of the Instrumentation

Formally, we define the instrumentation as follows:

Definition 3. *Let $\mathcal{P}$ be a program in the form defined in Table 1. Let $s_1, s_2, \ldots, s_n$ be the statement labels in $\mathcal{P}$. The instrumented program $\mathcal{P}'$ is obtained from $\mathcal{P}$ starting with a prolog of the form* **new**(ps_i) *where $i = 1, 2 \ldots n$. After the prolog, we rewrite $\mathcal{P}$ according to the translation rules shown in Table 2 and Table 3.*

Example 4. In the running example program (Figure 2), in $\mathcal{P}$, s_{11} writes *head.car* and in $\mathcal{P}'$, s_{11} writes *head.cdr.car.cdr*. This follows from:
$txe(head.car) = txa(head.car).cdr = txa(head).cdr.car.cdr = head.cdr.car.cdr$

3.4 Properties of the Instrumentation

In this section we show that the instrumentation has reduced the flow dependence problem to the may-alias problem. First the simulation of $\mathcal{P}$ by $\mathcal{P}'$ is shown in the Simulation Theorem. This implies that the instrumentation does not introduce any imprecision into the flow dependence analysis. We also show the Last Wrote Lemma which states that the instrumentation maintains the needed invariants. Because of the Simulation Theorem, and the Last Wrote Lemma, we are able to conclude that:

1. exactly all the flow dependences in $\mathcal{P}$ are found using a may-alias oracle on $\mathcal{P}'$.
2. using any conservative may-alias algorithm on $\mathcal{P}'$ always results in conservative flow dependences for $\mathcal{P}$.

Intuitively, by simulation, we mean that at every label of $\mathcal{P}$ and $\mathcal{P}'$, all the "observable properties" are preserved in $\mathcal{P}'$, given the same input. In our case, observable properties are:

- r-values printed by the write statements

	program *DestructiveAppend()*		**program** *InstrumentedDestructiveAppend()*
	begin		**begin**
			new(ps_i **)** $\forall i \in \{1, 2, \ldots, 12\}$
s_1:	**new(** *head* **)**	s_1:	**new(** *head* **)**
			head.car := ps_1
			new(*head.cdr* **)**
s_2:	**read(** *head.car* **)**	s_2:	**new(** *head.cdr.car* **)**
			head.cdr.car.car := ps_2
			read(*head.cdr.car.cdr)*
s_3:	*head.cdr* := **nil**	s_3:	*_rhs* := **nil**
			new(*head.cdr.cdr* **)**
			head.cdr.cdr.car := ps_3
			head.cdr.cdr.cdr := *_rhs*
s_4:	*tail* := *head*	s_4:	*_rhs* := *head.cdr*
			new(*tail* **)**
			tail.car := ps_4
			tail.cdr := *_rhs*
s_5:	**while(** *tail.car* $\neq$ 'x' **) do**	s_5:	**while(** *tail.cdr.car.cdr* $\neq$ 'x' **) do**

Fig. 2. The beginning of the example program and its corresponding instrumented program.

Table 2. The translation rules that define the instrumentation excluding the prolog. For simplicity, every assignment allocates a new instrumentation cons-cell. The variable *_rhs* is used as a temporary to store the right-hand side of an assignment to allow the same variable to be used on both sides of an assignment.

s_i: $\langle Access \rangle := \langle Exp \rangle$	$\Longrightarrow$	s_i: *_rhs* := $txe(\langle Exp \rangle)$ **new**$(txa(\langle Access \rangle))$ $txa(\langle Access \rangle).car := ps_i$ $txa(\langle Access \rangle).cdr := $ *_rhs*
s_i: **new**$(\langle Access \rangle)$	$\Longrightarrow$	s_i: **new**$(txa(\langle Access \rangle))$ $txa(\langle Access \rangle).car := ps_i$ **new**$(txa(\langle Access \rangle).cdr)$
s_i: **read**$(\langle Access \rangle)$	$\Longrightarrow$	s_i: **new**$(txa(\langle Access \rangle))$ $txa(\langle Access \rangle).car := ps_i$ **read**$(txa(\langle Access \rangle).cdr)$
s_i: **write**$(\langle Exp \rangle)$	$\Longrightarrow$	s_i: **write**$(txa(\langle Exp \rangle))$
$\langle Exp_1 \rangle = \langle Exp_2 \rangle$	$\Longrightarrow$	$txe(\langle Exp_1 \rangle) = txe(\langle Exp_2 \rangle)$
$\langle Exp_1 \rangle \neq \langle Exp_2 \rangle$	$\Longrightarrow$	$txe(\langle Exp_1 \rangle) \neq txe(\langle Exp_2 \rangle)$

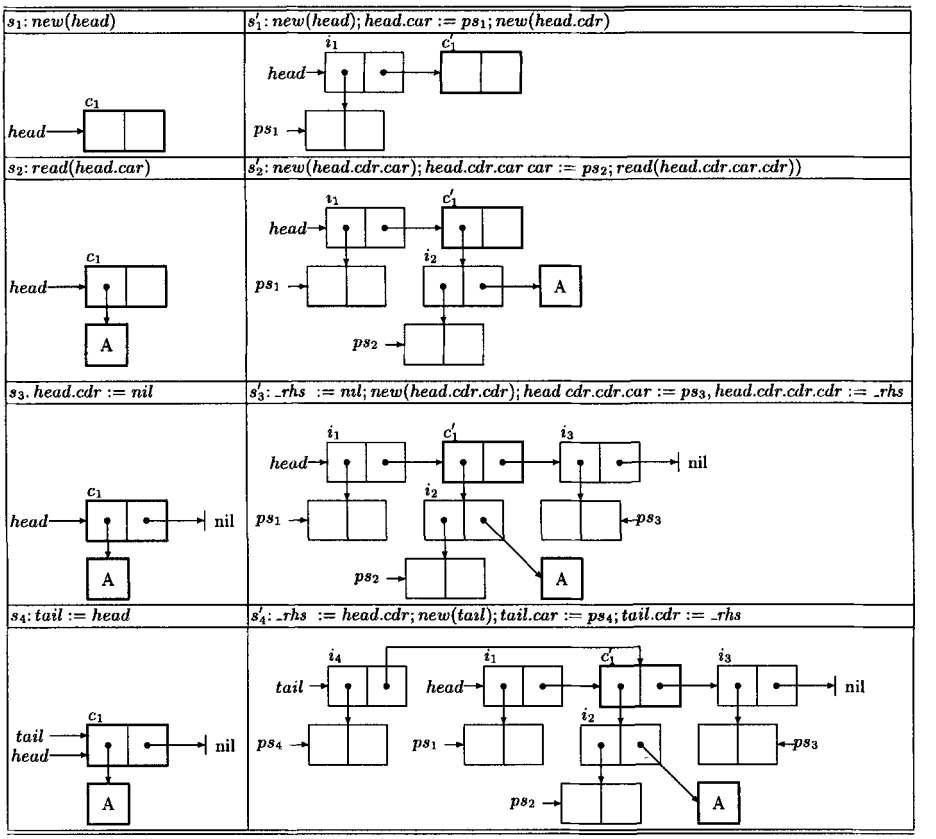

Fig. 3. The store of the original and the instrumented running example programs just before the loop on an input list starting with 'A'. For visual clarity, statement cons-cells not pointed to by an instrumentation cons-cell are not shown. Also, cons-cells are labeled and highlighted to show the correspondence between the stores of the original and instrumented programs.

Table 3. The function *txa* which maps an access path in the original program into the corresponding access path in the instrumented program. The function *txe* maps an expression into the corresponding expression in the instrumented program.

$txa(variable)$	$= variable$
$txa(\langle Access\rangle.\langle Sel\rangle)$	$= txa(\langle Access\rangle).cdr.\langle Sel\rangle$
$txe(\langle Access\rangle)$	$= txa(\langle Access\rangle).cdr$
$txe(atom)$	$= atom$
$txe(nil)$	$= nil$

– equalities of r-values

In particular, the execution sequences of $\mathcal{P}$ and $\mathcal{P}'$ at every label are the same. This discussion motivates the following definition:

Definition 5. *Let S be an arbitrary sequence of statement labels in $\mathcal{P}$ e_1, e_2 be expressions, and I be an input vector. We denote by $I, S \overset{\mathcal{P}}{\models} e_1 = e_2$ the fact that the input I causes S to be executed in $\mathcal{P}$, and in the store after S, the r-values of e_1 and e_2 are equal.*

Example 6. In the running example, *head.cdr.cdr* and *tail* denote the same cons-cell before the third iteration for inputs lengths of four or more. Therefore, $I, [s_1, s_2, s_3, s_4, s_5]([s_6, s_7, s_8, s_9, s_{10}])^2 \overset{\mathcal{P}}{\models} head.cdr.cdr = tail$

Theorem 7. (Simulation Theorem) *Given input I, expressions e_1 and e_2, and sequence of statement labels S:*

$$I, S \overset{\mathcal{P}}{\models} e_1 = e_2 \iff I, S \overset{\mathcal{P}'}{\models} txe(e_1) = txe(e_2)$$

Example 8. In the running example, before the first iteration, in the last box of Figure 3, *head* and *tail* denote the same cons-cell and *head.cdr* and *tail.cdr* denote the same cons-cell. Also, in the instrumented program, *head.cdr.cdr.cdr.cdr* and *tail.cdr* denote the same cons-cell before the third iteration for inputs of length four or more. Therefore, as expected from Example 6,
$I, [s_1, s_2, s_3, s_4, s_5]([s_6, s_7, s_8, s_9, s_{10}])^2 \overset{\mathcal{P}'}{\models} txe(head.cdr.cdr) = txe(tail)$

The utility of the instrumentation is captured in the following lemma.

Lemma 9. (Last Wrote Lemma) *Given input I, sequence of statement labels S, and an access path a, the input I leads to the execution of S in $\mathcal{P}$ in which the last statement that wrote into a is s_i if and only if*

$$I, S \overset{\mathcal{P}'}{\models} txa(a).car = ps_i.$$

Example 10. In the running example, before the first iteration, in the last box of Figure 3, we have $I, [s_1, s_2, s_3, s_4, s_5] \overset{\mathcal{P}'}{\models} txa(head).car = ps_1$ since s_1 is the statement that last wrote into *head*. Also, for input list $I = ['A', 'x']$, we have: $I, [s_1, s_2, s_3, s_4, s_5, s_{11}, s_{12}] \overset{\mathcal{P}'}{\models} txa(tail.car).car = ps_2$ since for such input s_2 last wrote into *tail.car* (through the assignment to *head.car*).

A single statement in our language can read from many memory locations. For example, in the running example program, statement s_5 reads from *tail* and *tail.car*. The complete read-sets for the statements in our language are shown in Tables 4 and 5.

We are now able to state the main result.

Table 4. Read-sets for the statements in our language.

s_i: $\langle Access \rangle := \langle Exp \rangle$	$rsa(\langle Access \rangle) \cup rse(\langle Exp \rangle)$
s_i: **new**($\langle Access \rangle$)	$rsa(\langle Access \rangle)$
s_i: **read**($\langle Access \rangle$)	$rse(\langle Access \rangle)$
s_i: **write**($\langle Exp \rangle$)	$rse(\langle Exp \rangle)$
$\langle Exp_1 \rangle = \langle Exp_2 \rangle$	$rse(\langle Exp_1 \rangle) \cup rse(\langle Exp_2 \rangle)$
$\langle Exp_1 \rangle \neq \langle Exp_2 \rangle$	$rse(\langle Exp_1 \rangle) \cup rse(\langle Exp_2 \rangle)$

Table 5. An inductive definition of rsa, the read-set for access-paths, and rse, the read-set for expressions.

$rsa(variable)$	$= \emptyset$
$rsa(\langle Access \rangle.\langle Sel \rangle)$	$= rsa(\langle Access \rangle) \cup \{\langle Access \rangle\}$
$rse(variable)$	$= \{variable\}$
$rse(\langle Access \rangle.\langle Sel \rangle)$	$= rse(\langle Access \rangle) \cup \{\langle Access \rangle.\langle Sel \rangle\}$
$rse(atom)$	$= \emptyset$
$rse(nil)$	$= \emptyset$

Theorem 11. (Flow Dependence Reduction) *Given program $\mathcal{P}$, its instrumented version $\mathcal{P}'$, and any two statement labels p and q. There is a flow dependence from p to q (in $\mathcal{P}$) if and only if there exists an access path, a, in the read-set of q, s.t. ps_p is a may-alias of $txa(a).car$ at q in $\mathcal{P}'$.*

Example 12. To find the flow dependences of s_5 in the running example:
s_5 : $while(tail.car \neq \text{`}x\text{'})$
First Tables 4 and 5 are used to determine the read-set of s_5:

$$rse(\langle Exp_1 \rangle) \cup rse(\langle Exp_2 \rangle) = rse(tail.car) \cup rse(\text{`}x\text{'}) = rse(tail) \cup \{tail.car\} \cup \emptyset$$
$$= \{tail\} \cup \{tail.car\} = \{tail, tail.car\}.$$

Then $txa(a).car$ is calculated for each a in the read-set:
$txa(tail).car = tail.car$
$txa(tail.car).car = txa(tail).cdr.car.car = tail.cdr.car.car$
Next the may-aliases to $tail.car$ and $tail.cdr.car.car$ are calculated by any may-aliases algorithm. Finally s_5 is flow dependent on the statements associated with the statement cons-cells that are among the may-aliases found to $tail.car$ and $tail.cdr.car.car$.

The Read-Sets and May-Aliases for the running example are summarized in Table 6.

From a complexity viewpoint our method can be very inexpensive. The program transformation time and space are linear in the size of the original program. In applying Theorem 11 the number of times the may-alias algorithm is invoked is also linear in the size of the original program, or more specifically, proportional

Table 6. Flow dependence analysis of the running example using a may-alias oracle.

Stmt	Read-Set	May-Aliases
s_1	$\emptyset$	$\emptyset$
s_2	$\{head\}$	$\{(head.car, ps_1)\}$
s_3	$\{head\}$	$\{(head.car, ps_1)\}$
s_4	$\{head\}$	$\{(head.car, ps_1)\}$
s_5	$\{tail, tail.car\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}),$ $(tail.cdr.car.car, ps_2), (tail.cdr.car.car, ps_7)\}$
s_6	$\emptyset$	$\emptyset$
s_7	$\{temp\}$	$\{(temp.car, ps_6)\}$
s_8	$\{temp\}$	$\{(temp.car, ps_6)\}$
s_9	$\{tail, temp\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}), (temp.car, ps_6)\}$
s_{10}	$\{tail, tail.cdr\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}), (tail.cdr.cdr.car, ps_9)\}$
s_{11}	$\{head, head.car\}$	$\{(head.car, ps_1), (head.cdr.car.car, ps_2)\}$
s_{12}	$\{tail, tail.car\}$	$\{(tail.car, ps_4), (tail.car, ps_{10}),$ $(tail.cdr.car.car, ps_2), (tail.cdr.car.car, ps_7)\}$

to the size of the read-sets. It is most likely that the complexity of the may-alias algorithm itself is the dominant cost.

4 Plug and Play

An important corollary of Theorem 11 is that an arbitrary conservative may-alias algorithm on $\mathcal{P}'$ yields a conservative solution to the flow dependence problem on $\mathcal{P}$. Since existing may-alias algorithms often yield results which are difficult to compare, it is instructive to consider the flow dependences computed by various algorithms on the running example program.

- The algorithm of [9] yields the may-aliases shown in column 3 of Table 6. Therefore, on this program, this algorithm yields the exact flow dependences shown in Figure 1.
- The more efficient may-alias algorithms of [23, 14, 30, 29] are useful to find flow dependences in programs with disjoint data structures. However, in programs with recursive data structures such as the running example, they normally yield many superfluous may-aliases leading to superfluous flow dependences. For example, [23] is not able to identify that *tail* points to an acyclic list. Therefore, it yields that *head.car* and ps_7 are may-aliases at s_{11}. Therefore, it will conclude that the value of *head.car* read at s_{11} may be written inside the loop (at statement s_7).
- The algorithm of [27] finds, in addition to the correct dependences, superfluous flow dependences in the running example. For example, it finds that s_5 has a flow dependence on s_8. This inaccuracy is attributable to the anonymous nature of the second cons-cell allocated with each new statement. There are two possible ways to remedy this inaccuracy:

- Modify the algorithm so that it is 2-bounded, i.e., also keeps track of *car* and *cdr* fields of variables. Indeed, this may be an adequate solution for general k-bounded approaches, e.g., [19] by increasing k to $2k$.
- Modify the transformation to assign unique names to these cons-cells. We have implemented this solution, and tested it using the PAG [3] implementation of the [27] algorithm[3] and found exactly all the flow dependences in the running example.

5 Conclusions

In this paper, we showed that may-alias algorithms can be used, without any modification, to compute program dependences. We are hoping that this will lead to more research in finding practical may-alias algorithms to compute good approximations for flow dependences.

For simplicity, we did not optimize the memory usage of the instrumented program. In particular, for every executed instance of a statement in the original program that writes to the store, the instrumented program creates a new instrumentation cons-cell. This extra memory usage is harmless to may-alias algorithms (for some algorithms it can even improve the accuracy of the analysis, e.g., [5]). In cases where the instrumented program is intended to be executed, it is possible to drastically reduce the memory usage through cons-cell reuse.

Finally, it is worthwhile to note that flow dependences can be also used to find may-aliases. Therefore, may-aliases are necessary in order to compute flow dependences. For example, Figure 4 contains a program fragment that provides the instrumentation to "check" if two program variables v_1 and v_2 are may aliases at program point p. This instrumentation preserves the meaning of the original program and has the property that v_1 and v_2 are may-aliases at p if and only if s_2 has a flow dependence on s_1.

p: **if** $v_1 \neq nil$ **then**
  s_1: $v_1.cdr := v_1.cdr$ **fi**
  if $v_2 \neq nil$ **then**
    s_2: $write(v_2.cdr)$ **fi**

Fig. 4. A program fragment such that v_1 and v_2 are may-aliases at p if and only if s_2 has a flow dependence on s_1.

Acknowledgments

We are grateful for the helpful comments of Thomas Ball, Michael Benedikt, Thomas Reps, and Reinhard Wilhelm for their comments that led to substantial

[3] On a SPARCstation 20, PAG used 0.53 seconds of cpu time.

improvements of this paper. We also would like to thank Martin Alt and Florian Martin for PAG, and their PAG implementation of [27] for a C subset.

References

1. H. Agrawal and J.R. Horgan. Dynamic program slicing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, volume 25 of *ACM SIGPLAN Notices*, pages 246–256, White Plains, New York, June 1990.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.
3. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 33–50. Springer-Verlag, 1995.
4. J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *ACM Symposium on Principles of Programming Languages*, pages 29–41, New York, NY, 1979. ACM Press.
5. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.
6. J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, New York, NY, 1993. ACM Press.
7. K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 57–66, New York, NY, 1988. ACM Press.
8. A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.
9. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 230–241, New York, NY, 1994. ACM Press.
10. A. Deutsch. Semantic models and abstract interpretation for inductive data structures and pointers. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*, pages 226–228, New York, NY, June 1995. ACM Press.
11. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Languages Design and Implementation*, New York, NY, 1994. ACM Press.
12. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 3(9):319–349, 1987.
13. R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1996. ACM Press.
14. R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for c. In *Proc. of the 8th Intl. Work. on Languages and Compilers for Parallel Computing*, number 1033 in Lecture Notes in Computer Science, pages 515–534, Columbus, Ohio, August 1995. Springer-Verlag.

15. R. Ghiya and L.J. Hendren. Putting pointer analysis to work. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1998.
16. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, Ithaca, N.Y., Jan 1990.
17. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.
18. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*, volume 24 of *ACM SIGPLAN Notices*, pages 28–40, Portland, Oregon, June 1989. ACM Press.
19. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
20. N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages*, pages 66–74, New York, NY, 1982. ACM Press.
21. D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, pages 207–218, New York, NY, 1981. ACM Press.
22. W. Land, B.G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 56–67, 1993.
23. W. Landi and B.G. Ryder. Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*, pages 93–103, New York, NY, January 1991. ACM Press.
24. J.R. Larus. Refining and classifying data dependences. Unpublished extended abstract, Berkeley, CA, November 1988.
25. J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 21–34, New York, NY, 1988. ACM Press.
26. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, New York, NY, 1984. ACM Press.
27. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Symposium on Principles of Programming Languages*, New York, NY, January 1996. ACM Press.
28. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 1997. To Appear.
29. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*, 1997.
30. B. Steengaard. Points-to analysis in linear time. In *ACM Symposium on Principles of Programming Languages*. ACM, New York, January 1996.
31. R.P. Willson and M.S. Lam. Efficient context-sensitive pointer analysis for c programs. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 1–12, La Jolla, CA, June 18-21 1995. ACM Press.