Nordic Journal of Computing 5(1998), 361–386.

BUILDING A BRIDGE BETWEEN POINTER ALIASES AND PROGRAM DEPENDENCES

JOHN L. ROSS
University of Chicago, Department of Computer Science
1100 East 58th Street, Chicago, IL 60637, USA
johnross@cs.uchicago.edu

MOOLY SAGIV
Tel-Aviv University, Department of Computer Science
School of Mathematical Sciences, Tel-Aviv 69978, Israel
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 reduction 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.

CR Classification: D.3.3, D.3.4

Key words: alias analysis, dataflow analysis, pointer analysis, program dependences, static analysis

1. Introduction

It is well known that programs with pointers are hard to understand, to debug, and to optimize. In recent years many interesting algorithms to conservatively analyze programs with pointers have been published. Roughly speaking, these algorithms [22, 23, 28, 6, 17, 18, 25, 9, 7, 10, 14, 15, 33] conservatively (safely) 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 usually insufficient for compiler optimizations, automatic code parallelizations, instruction scheduling for super-

Received August 1998; revised October 1998; accepted October 1998.

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 [24, 29, 13]).

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

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

Fig. 1 shows the scheme of our approach.

To see why the problem of computing flow dependences from may-aliases is not trivial, consider the example program fragment shown in Fig. 2(a). A formal definition of flow dependences is given in Def. 1. For the purposes of this discussion, a statement $\mathtt{l}_2$ *depends on* $\mathtt{l}_1$ if the value produced in $\mathtt{l}_1$ is directly used at $\mathtt{l}_2$. The naive solution is to conclude that in Fig. 2(a) $\mathtt{l}_2$ depends on $\mathtt{l}_1$ if and only if `p` and `q` are may-aliases at $\mathtt{l}_2$, i.e., they can refer to the same location at $\mathtt{l}_2$. Fig. 2(b) shows that the naive solution can sometimes be too conservative, when statement $\mathtt{l}_{1.5}$ overwrites the location pointed to by `p`. Fig. 2(c) shows that the naive solution may even miss the dependency between $\mathtt{l}_1$ and $\mathtt{l}_2$, when statement $\mathtt{l}_{1.5}$ overwrites `p`.

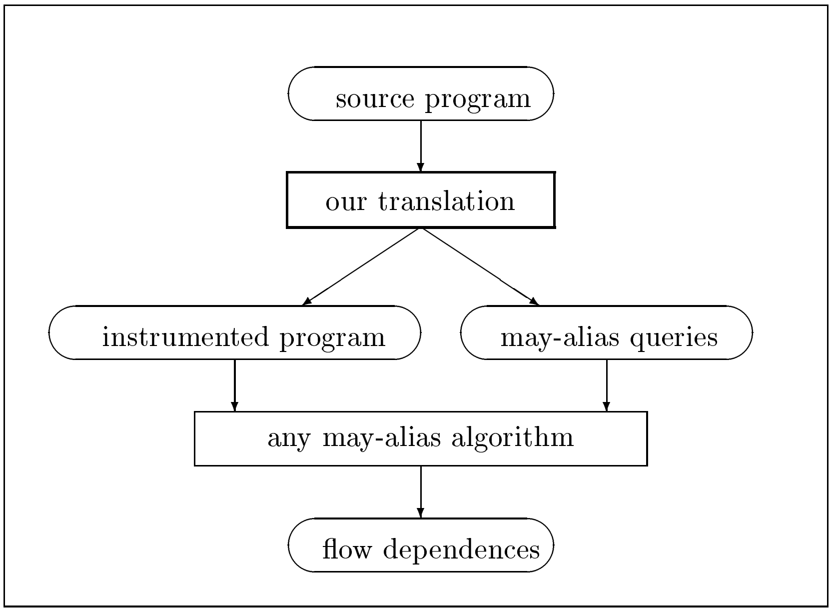


Fig. 1: Schematic of our method for calculating flow dependences using a may-alias algorithm.

int y	int y, a	int y, a, b
int *p, *q	int *p, *q, *t	int *p, *q
	q = & a	q = & a
	p = q	p = q
	t= p	
l_1: *p = 5	l_1: *p = 5	l_1: *p = 5
$l_{1.5}$:	$l_{1.5}$: *t = 7	$l_{1.5}$: p = &b
l_2: y = *q	l_2: y = *q	l_2: y = *q
(a)	(b)	(c)

Fig. 2: A motivating example to demonstrate the differences between may-aliases and flow dependences.

In this paper, we reduce the program dependence problem, a problem of great practical importance, to the may-alias problem, a problem with many competing solutions. Of course, different may-alias algorithms have their own strengths, weaknesses, and computational costs. The reduction has the property that as long as the may-alias algorithm is conservative, the dependences computed are 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 directly 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:

- We unify the concepts of program dependences and may-aliases. While these concepts are seemingly different, we provide linear time and space reductions between them. Thus may-aliases can be used to find program dependences, and program dependences can be used to find may-aliases.
- We provide an answer to the previously open question about the ability to use "storeless" (see [9, 10, 11]) may-alias algorithms such as [10, 33] to find dependences.

  One of the simplest storeless may-alias algorithm is due to Ghiya and Hendren [14]. In [16], 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. This becomes more interesting when considering more sophisticated algorithms, such as [6, 9, 10, 33], which are sometimes able to infer the shape of recursive data structures. Furthermore, we believe that our reduction is actually simpler to understand than the names introduced in [16] because we are proving the correctness of a program transformation instead of modifying a particular approximation algorithm.

- Our limited experience with the reduction indicates that storeless may-alias algorithms such as [10, 33] yield quite precise dependence information (See Section 4). For example, Fig. 5 contains a program that builds a list and the instrumented program that results from our construction. When the algorithm of [10] is applied to the instrumented program, it locates the exact program dependences (shown in Fig. 3).

  It is an open question if the flow-sensitive algorithms of [10, 33] or even simpler flow-sensitive algorithms such as [14, 16, 19] can scale for large programs. The main difficulty is the explosion in space, which may be quadratic in the size of the analyzed program.

  Fortunately, in Section 4, we show that a variant of our instrumentation may be used to calculate useful flow dependences even when used with a flow-insensitive algorithm. More specifically, the flow-insensitive algorithm of Yong, Horwitz, and Reps [37] calculates dependences in the program shown in Fig. 3, with only one false dependence (between $\mathtt{l}_8$ and $\mathtt{l}_{10}$). This algorithm is a version of Andersen's pointer analysis [4] that has been refined to differentiate among fields of structures. This is very encouraging from an implementation perspective.

- Our approach provides a method to compare the time and precision of different may-alias algorithms, namely 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 [25, 12, 36, 35, 34].

Our program instrumentation closely resembles the "instrumented semantics" of Horwitz, Pfeiffer, and Reps [20]. They propose to change the program semantics so that the interpreter will carry around program statements. We instrument the program itself to record statement information locally. 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 [22] (which is imprecise and doubly exponential in space).

An additional benefit of our shift from semantic 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 [5, 8, 26, 7] with reaching definitions [2] or by combining conflict analysis [28] with reaching definitions as done in [27]. 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 [7, Section 6.2] for an interesting discussion). Also, the computation of reaching definitions is costly.

1.2 Outline of the rest of the paper

In Section 2.1, we describe a simple Lisp-like language that is used throughout the 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 a record of the statement 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

A program in our simplified language (which follows [22, 6]) is a sequence of labeled statements, as defined in Table I. For brevity, we only allow low-level (conditional) branches. It is easy to extend the language to handle high-level control flow constructs and procedures. Also, to simplify the exposition we assume, without loss of generality, that the program statements are labeled from 1 to n. These labels are used in `goto` statements, the definitions of aliases and dependences, and the instrumentation. Lisp-like memory access and explicit destructive assignment statements are allowed. Memory access paths are represented by $\langle a \rangle$ and expressions are represented by $\langle e \rangle$.

TABLE I: A simplified language with dynamic memory and destructive updates.

$\langle Lst\rangle ::= \texttt{l}{:}\langle St\rangle$
$\langle St\rangle ::= \langle a\rangle{:=}\langle e\rangle \mid \texttt{new}(\langle a\rangle) \mid \texttt{read}(\langle a\rangle) \mid \texttt{write}(\langle e\rangle)$
$\langle St\rangle ::= \texttt{if}\ \langle e\rangle = \langle e\rangle\ \texttt{goto l} \mid \texttt{goto l}$
$\langle a\rangle ::= v \mid \langle a\rangle.\langle Sel\rangle$
$\langle e\rangle ::= \langle a\rangle \mid atom \mid \texttt{nil}$
$\langle Sel\rangle ::= \texttt{car} \mid \texttt{cdr}$

Fig. 3 shows a program that will be used throughout the paper as a running example. This program reads atoms and builds them into a list by successively performing a destructive update on the cdr field of the tail of the list.

```
  program Append()
 l1:  new(head)
 l2:  read(head.car)
 l3:  head.cdr:=nil
 l4:  tail:=head
 l5:  if tail.car = 'x' goto l12
 l6:    new(temp)
 l7:    read(temp.car)
 l8:    temp.cdr:=nil
 l9:    tail.cdr:=temp
 l10:   tail:=tail.cdr
 l11:   goto l5
 l12: write(head.car)
 l13: write(tail.car)
```

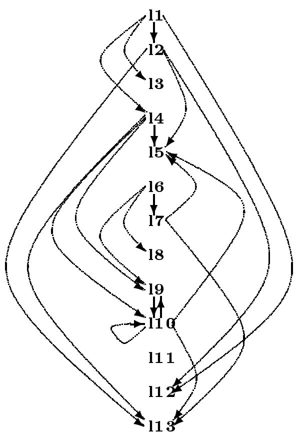


Fig. 3: A program that builds a list by destructively appending elements to `tail`. The graph on the right shows the program's flow dependences.

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) [24, 13]. In this paper, we focus on flow dependences between program statements.

A single statement in our language can read from many memory locations. For example, in the running example program, statement $\texttt{l}_5$ reads from `tail` and `tail.car`. The read-sets and write-sets for the statements in our language are shown in Tables II and III. All proper prefixes of an access path are read when the access path occurs in a defining position. The full

TABLE II: The access paths that are read and written by the statements in our language.

statement	*readSet*(statement)	*writeSet*(statement)
$\langle a\rangle := \langle e\rangle$	$readSet_L(\langle a\rangle) \cup readSet_R(\langle e\rangle)$	$writeSet_L(\langle a\rangle)$
`new`($\langle a\rangle$)	$readSet_L(\langle a\rangle)$	$writeSet_L(\langle a\rangle)$
`read`($\langle a\rangle$)	$readSet_L(\langle a\rangle)$	$writeSet_L(\langle a\rangle)$
`write`($\langle e\rangle$)	$readSet_R(\langle e\rangle)$	$\emptyset$
`if` $\langle e_1\rangle = \langle e_2\rangle$ `goto l`	$readSet_R(\langle e_1\rangle) \cup readSet_R(\langle e_2\rangle)$	$\emptyset$
`goto l`	$\emptyset$	$\emptyset$

TABLE III: An inductive definition of the read sets $readSet_L$ and $writeSet_L$ for left-hand-side access-paths, and $readSet_R$ for right-hand-side expressions.

access-path	$readSet_{\mathrm{L}}$(access-path)	$writeSet_{\mathrm{L}}$(access-path)
v	$\emptyset$	$\{ v\}$
$\langle a\rangle.\langle Sel\rangle$	$readSet_L(\langle a\rangle) \cup \{\langle a\rangle\}$	$\{\langle a\rangle.\langle Sel\rangle\}$
expression	$readSet_{\mathrm{R}}$(expression)	
v	$\{v\}$	
$\langle a\rangle.\langle Sel\rangle$	$readSet_R(\langle a\rangle) \cup \{\langle a\rangle.\langle Sel\rangle\}$	
atom	$\emptyset$	
nil	$\emptyset$	

access path is read when it occurs in a using position. It is written when it occurs on a defining position.

Our language allows programs to explicitly modify their store through pointers. Because of this, we phrase the definition of flow dependence in terms of memory *locations* (cons-cells) and not variable names. We shall follow [20] in defining flow dependence as:

DEFINITION 1. (FLOW DEPENDENCE) *Consider labeled statements* $\mathtt{l_i}$: $\mathtt{st_i}$ *and* $\mathtt{l_j}$: $\mathtt{st_j}$. *We say that* $\mathtt{l_i}$ has a flow dependence on $\mathtt{l_j}$, *if* $\mathtt{st_j}$ *writes into memory location, loc, that* $\mathtt{st_i}$ *reads, and there is no intervening write into loc along an execution path by which* $\mathtt{l_i}$ *is reached from* $\mathtt{l_j}$.

Notice that in this definition the location of a variable is its l-value.

Fig. 3 shows the flow dependences for the running example program. Notice that $\mathtt{l}_{12}$ is flow dependent on only $\mathtt{l}_1$ and $\mathtt{l}_2$, while $\mathtt{l}_{13}$ is flow dependent on $\mathtt{l}_2$, $\mathtt{l}_4$, $\mathtt{l}_7$, and $\mathtt{l}_{10}$. This information could be used by slicing tools to find that the loop need not be executed to print `head.car` in $\mathtt{l}_{12}$, or by an instruction scheduler to reschedule $\mathtt{l}_{12}$ for anytime after $\mathtt{l}_2$. Also, $\mathtt{l}_3$, $\mathtt{l}_8$, and $\mathtt{l}_{11}$ have no statements dependent on them, making them candidates for elimination. Thus, even in this simple example, knowing the flow dependences potentially would allow several code transformations.

Because determining the exact flow dependences in an arbitrary program is undecidable, approximation algorithms must be used.

DEFINITION 2. *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 at a given program point, two access paths could denote the same cons-cell.

DEFINITION 3. (MAY-ALIAS) Two access-paths are may-aliases at a label $\mathtt{l_i}$ *in a program, if there exists a path to* $\mathtt{l_i}$ *where both denote the same location (cons-cell).*

In the running example program, `head.cdr.cdr` and `tail` are may-aliases at $\mathtt{l_6}$ because just before the third iteration these access paths denote the same cons-cell. However, `tail.cdr.cdr` is not a may-alias to `head` because they can never denote the same cons-cell.

Because the may-alias problem is also undecidable, approximation algorithms must be used.

DEFINITION 4. *A may-alias approximation algorithm is* conservative *if it always finds a superset of the true may-aliases.*

3. The instrumentation transformation

In this section, the instrumentation transformation is defined. For notational simplicity, $\mathcal{P}$ stands for an arbitrary fixed program, and $ins(\mathcal{P})$ stands for its instrumented version. This section is organized as follows: In Section 3.1, we briefly sketch the main ideas behind the instrumentation transformation. Then, in Section 3.2, we show the instrumentation of the running example program, and its affect on a given input. Then, in Section 3.3, we formally define the translation. Finally, in Section 3.4, we show that may-aliases of $ins(\mathcal{P})$ precisely determine the flow dependences in $\mathcal{P}$.

3.1 The main idea behind the instrumentation transformation

In this section, we explain the main idea behind the instrumentation scheme. Fig. 4 shows a tiny program, $\mathcal{P}$, and the corresponding instrumented program, $ins(\mathcal{P})$. We also show the stores that arise when these two programs are executed. $ins(\mathcal{P})$ simulates the values and the execution sequences of $\mathcal{P}$ and, in addition, records for every variable `v` the statement from $\mathcal{P}$ that last wrote into `v`. This instrumentation information is recorded in `v.car` (while

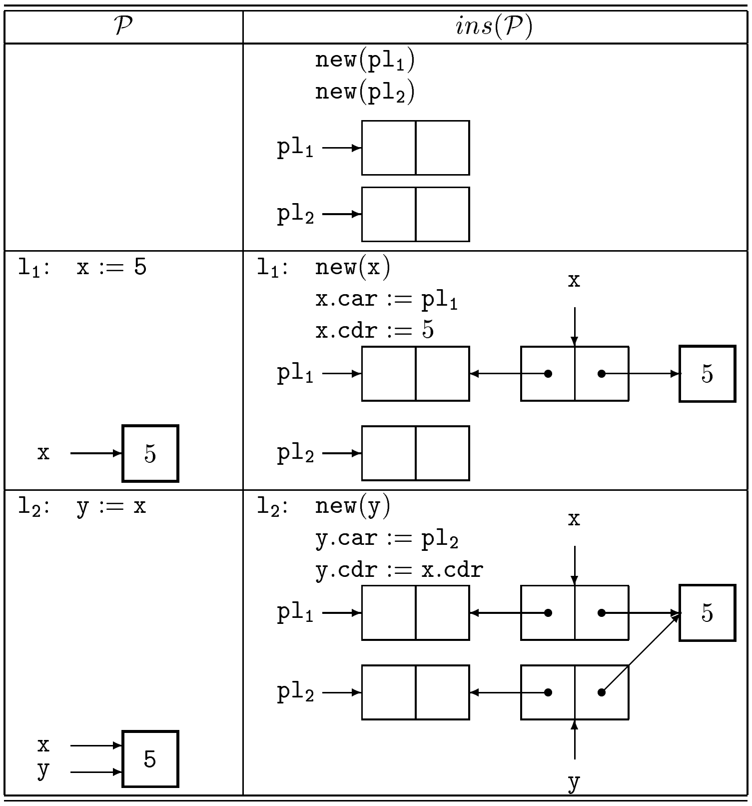


Fig. 4: A tiny program that illustrates the main ideas used in the instrumentation scheme.

storing the original contents of `v` in `v.cdr`). This "totally static" instrumentation[1] allows program dependences to be recovered from may-alias queries on $ins(\mathcal{P})$.

The instrumented program, $ins(\mathcal{P})$, starts by allocating two new cons-cells, which are pointed to by two new program variables, $\mathtt{pl}_1$ and $\mathtt{pl}_2$. We refer to these cons-cells as the *statement-cells*. The usage of these cells will become apparent shortly. The rest of $ins(\mathcal{P})$ contains a statement block starting at $\mathtt{l_i}$ for every labeled statement $\mathtt{l_i}\colon\langle St\rangle$ in $\mathcal{P}$. For instance, in $\mathcal{P}$, the first statement sets `x` to 5. Therefore, the instrumented program performs the following three store operations:

(1) Allocates a new cons-cell, referred to as an *instrumentation-cell*, and sets `x` to point to it.

(2) Records the statement that (in $\mathcal{P}$) last wrote into `x`. This is done by setting the car field of the instrumentation-cell to point to the

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

statement-cell that is pointed to by $\mathtt{pl}_1$ (via the assignment $\mathtt{x.car} := \mathtt{pl}_1$).

(3) Records the value that x would have in $\mathcal{P}$. This is done by assigning the value of x in $\mathcal{P}$ to the cdr field of the instrumentation-cell (via the assignment $\mathtt{x.cdr} := 5$).

In general, there is a flow dependence, in $\mathcal{P}$, from a given statement $\mathtt{l_i}$ to $\mathtt{l_j}$: $y := x$, if and only if `x.car` and $\mathtt{pl_i}$ may refer to the same statement-cell in $ins(\mathcal{P})$. For example, in Fig. 4, there is a flow dependence from $\mathtt{l}_1$ to $\mathtt{l}_2$ because `x.car` and $\mathtt{pl}_1$ may refer to the same location. Therefore, we can recover the flow dependences of $\mathcal{P}$ by querying the may-aliases of "used" access paths.

3.2 The instrumentation of the running example

On a slightly larger scale, Fig. 5 shows the running example program and its instrumented version. Fig. 6 shows the stores of both of 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-cell for each statement in the original program (not shown). Then, for every statement in the original program, the instrumented statement block in the instrumented program records the appropriate value and, for every (original) location written, the last statement that wrote into it. The variable `instV` is used as a temporary to point to the last allocated instrumented cell.

Let us now illustrate this for the statements $\mathtt{l}_1$ through $\mathtt{l}_4$ in Fig. 6.

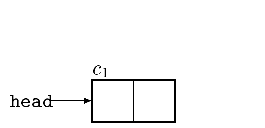
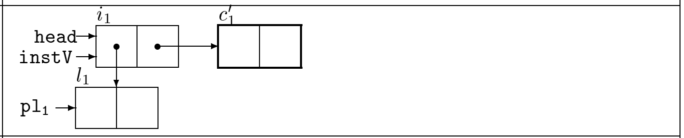
- In the original program, after $\mathtt{l}_1$, `head` points a new uninitialized cell, c_1. In the instrumented program, after the block of statements $\mathtt{l}_1$, `head` points to an instrumentation-cell, i_1, `head.car` points to the statement-cell for $\mathtt{l}_1$, and `head.cdr` points to c'_1.
- In the original program, after l_2, `head.car` points to the atom A. In the instrumented program, after the block of statements labeled by $\mathtt{l}_2$, `head.cdr.car` points to an instrumentation-cell, i_2, `head.cdr.car.car` points to the statement-cell for $\mathtt{l}_2$, and `head.cdr.car.cdr` points to A.
- In the original program, after $\mathtt{l}_3$, `head.cdr` points to `nil`. In the instrumented program, after the block of statements labeled by $\mathtt{l}_3$, `head.cdr.cdr` points to an instrumentation-cell, i_3, `head.cdr.cdr.car` points to the statement-cell for $\mathtt{l}_3$, and `head.cdr.cdr.cdr` points to `nil`.
- In the original program, after $\mathtt{l}_4$, `tail` points to the cell c_1. In the instrumented program, after the block of statements labeled by $\mathtt{l}_4$, `tail` points to an instrumentation-cell, i_4, `tail.car` points to the statement-cell for $\mathtt{l}_4$, and `tail.cdr` points to c'_1. Notice how the sharing (the aliasing) of the values of `head` and `tail` in $\mathcal{P}$ is preserved by the transformation (i.e., `head.cdr` and `tail.cdr` both point to c'_1).

```
  program Append()

l1:  new(head)




l2:  read(head.car)




l3:  head.cdr:=nil




l4:  tail :=head




l5:  if tail.car = 'x'
     goto l12
l6:    new(temp)




l7:    read(temp.car)




l8:    temp.cdr:=nil




l9:    tail.cdr:=temp




l10:   tail:=tail.cdr




l11:   goto l5
l12: write(head.car)
l13: write(tail.car)
```

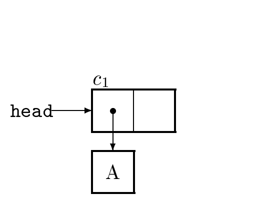
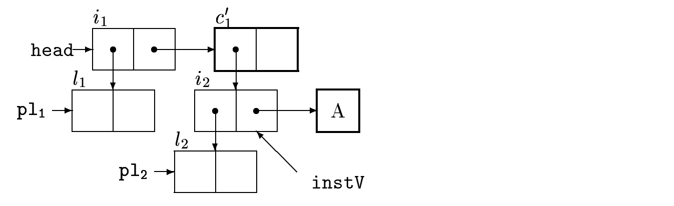
```
  program InsAppend()
     new(pli) ∀i ∈ {1,2,...,13}
l1:  new(instV)
     instV.car:=pl1
     new(instV.cdr)
     head:=instV
l2:  new(instV)
     instV.car:=pl2
     read(instV.cdr)
     head.cdr.car:=instV
l3:  new(instV)
     instV.car:=pl3
     instV.cdr:=nil
     head.cdr.cdr:=instV
l4:  new(instV)
     instV.car:=pl4
     instV.cdr:=head.cdr
     tail:=instV
l5:  if tail.cdr.car.cdr = 'x'
     goto l12
l6:    new(instV)
       instV.car:=pl6
       new(instV.cdr)
       temp:=instV
l7:    new(instV)
       instV.car:=l7
       read(instV.cdr)
       temp.cdr.car:=instV
l8:    new(instV)
       instV.car:=pl8
       instV.cdr:=nil
       temp.cdr.cdr:=instV
l9:    new(instV)
       instV.car:=pl9
       instV.cdr:=temp.cdr
       tail.cdr.cdr:=instV
l10:   new(instV)
       instV.car:=pl10
       instV.cdr:=tail.cdr.cdr.cdr
       tail.cdr:=instV
l11:   goto l5
l12: write(head.cdr.car.cdr)
l13: write(tail.cdr.car.cdr)
```

Fig. 5: A program that builds a list by destructively appending elements and the corresponding instrumented program.

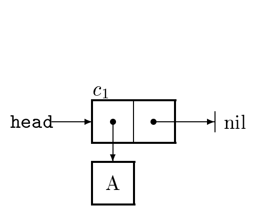
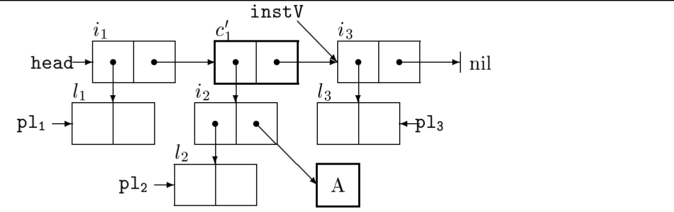
$\mathtt{l}_1$: `new(head)`	$\mathtt{l}_1$: `new(instV)`
	`instV.car := pl`$_1$
	`new(instV.cdr)`
	`head := instV`

$\mathtt{l}_2$: *read*(*head*.*car*)	$\mathtt{l}_2$: `new(instV)`
	`instV.car:=pl`$_2$
	`read(instV.cdr)`
	`head.cdr.car := instV`

$\mathtt{l}_3$: `head.cdr := nil`	$\mathtt{l}_3$: `new(instV)`
	`instV.car := pl`$_3$
	`instV.cdr := nil`
	`head.cdr.cdr := instV`

$\mathtt{l}_4$: `tail := head`	$\mathtt{l}_4$: `new(instV)`
	`instV.car := pl`$_4$
	`instV.cdr:= head.cdr`
	`tail := instV`

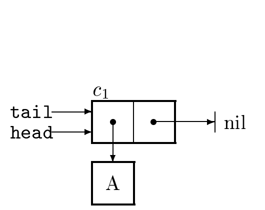
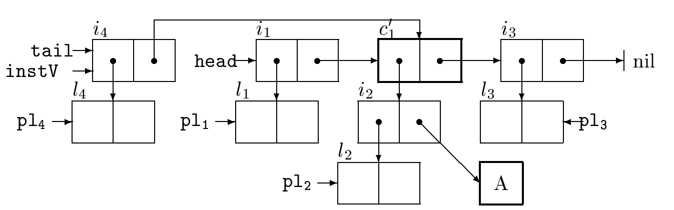
	

Fig. 6: The store of the original and the instrumented running example programs just before the loop on an input list starting with 'A'. For visual clarity, statement cells not pointed to by an instrumentation cell are not shown. Also, cells are labeled and highlighted to show the correspondence between the stores of the original and instrumented programs.

3.3 A formal definition of the instrumentation transformation

Formally, the instrumented program is defined as follows

DEFINITION 5. (THE INSTRUMENTED PROGRAM) *Let $\mathcal{P}$ be a program of the form:*

$$\begin{array}{l} \mathtt{l_1}\colon \mathtt{st_1} \\ \mathtt{l_2}\colon \mathtt{st_2} \\ \vdots \\ \mathtt{l_n}\colon \mathtt{st_n} \end{array}$$

where the statements $\mathtt{st_i}$ *are of the form defined in Table I.*

The instrumented program $ins(\mathcal{P})$ is defined by:

$$\begin{array}{l} \mathtt{new(pl_1)} \\ \mathtt{new(pl_2)} \\ \vdots \\ \mathtt{new(pl_n)} \\ ins_S(\mathtt{l_1}{:}st_1) \\ ins_S(\mathtt{l_2}{:}st_2) \\ \vdots \\ ins_S(\mathtt{l_n}{:}st_n) \end{array}$$

where ins_S is defined in Table IV and Table V.

TABLE IV: The function ins_S that defines the instrumentation of a statement. The variable `instV` is a temporary that points to new instrumentation-cells. The function ins_L returns the corresponding l-value, in $ins(\mathcal{P})$, for a given access path in $\mathcal{P}$. The function ins_R returns the corresponding r-value for a given expression in $\mathcal{P}$. These additional functions are defined in Table V.

$\langle$labeled statement$\rangle$	$\mathrm{ins_S}(\langle$labeled statement$\rangle)$
$\mathtt{l_i}$: $\langle a\rangle := \langle e\rangle$	$\mathtt{l_i}$: `new(instV)` `instV.car` := $\mathtt{pl_i}$ `instV.cdr` :=$ins_R(\langle e\rangle)$ $ins_L(\langle a\rangle)$:= `instV`
$\mathtt{l_i}$: `new(`$\langle a\rangle$`)`	$\mathtt{l_i}$: `new(instV)` `instV.car` := $\mathtt{pl_i}$ `new(instV.cdr)` $ins_L(\langle a\rangle)$:= `instV`
$\mathtt{l_i}$: `read(`$\langle a\rangle$`)`	$\mathtt{l_i}$: `new(instV)` `instV.car` := $\mathtt{pl_i}$ `read(instV.cdr)` $ins_L(\langle a\rangle)$:= `instV`
$\mathtt{l_i}$: `write(`$\langle e\rangle$`)`	$\mathtt{l_i}$: `write(`$ins_R(\langle e\rangle)$`)`
$\mathtt{l_i}$: `if` $\langle e_1\rangle = \langle e_2\rangle$ `goto l`	$\mathtt{l_i}$: `if` $ins_R(\langle e_1\rangle) = ins_R(\langle e_2\rangle)$ `goto l`
$\mathtt{l_i}$: `goto l`	$\mathtt{l_i}$: `goto l`

TABLE V: The functions ins_L and ins_R that navigate through the stores produced by instrumented programs. The function ins_L returns the corresponding l-value, in $ins(\mathcal{P})$, for a given access path in $\mathcal{P}$. The function ins_R returns the corresponding r-value for a given expression.

access-path	ins_L(access-path)
v	v
$\langle a\rangle.\langle Sel\rangle$	$ins_L(\langle a\rangle).\mathtt{cdr}.\langle Sel\rangle$
expression	ins_R(expression)
$\langle a\rangle$	$ins_L(\langle a\rangle).\mathtt{cdr}$
`at`	`at`
`nil`	`nil`

EXAMPLE 1. Consider the running example program $\mathcal{P}$ shown in Fig. 5 and the statement $\mathtt{l}_{12}$. We have:

$$
\begin{aligned}
& ins_S(\mathtt{l}_{12}\text{: }\mathtt{write}(\mathtt{head.car})) \\
&\quad = \mathtt{l}_{12}\text{: }\mathtt{write}(ins_R(\mathtt{head.car})) \\
&\quad = \mathtt{l}_{12}\text{: }\mathtt{write}(ins_L(\mathtt{head.car}).\mathtt{cdr}) \\
&\quad = \mathtt{l}_{12}\text{: }\mathtt{write}(ins_L(\mathtt{head}).\mathtt{cdr.car.cdr}) \\
&\quad = \mathtt{l}_{12}\text{: }\mathtt{write}(\mathtt{head.cdr.car.cdr})
\end{aligned}
$$

Therefore, in $ins(\mathcal{P})$, the statement at $\mathtt{l}_{12}$ outputs the value of the expression `head.cdr.car.cdr`.

3.4 Properties of the instrumentation

In this section, we show that the instrumentation transformation has reduced the flow dependence problem to the may-alias problem. First, the simulation of $\mathcal{P}$ by $ins(\mathcal{P})$ is shown in the Simulation Theorem. This implies that the transformation does not introduce any imprecision into the flow-dependence analysis. We also show the Last Wrote Lemma, which states that the transformation maintains the needed invariants regarding which statement last wrote into each memory location. 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 $ins(\mathcal{P})$.

(2) using any conservative may-alias algorithm on $ins(\mathcal{P})$ always results in conservative flow dependences for $\mathcal{P}$.

3.4.1 The Simulation Theorem

In order to formulate precisely the Simulation Theorem, we now formally define the meaning of programs in our simplified language. We start by defining two-level stores.

DEFINITION 6. (TWO-LEVEL STORE) *A (two-level)* store *is a quadruple* $\langle L, env, car, cdr\rangle$*, where:*

- L *is a finite set of* locations*. Atoms is a possibly infinite set of basic* atoms*. We assume that L and Atoms are disjoint. For simplicity, we also assume that there exists a distinguished element* $\perp$ *which stands for the* uninitialized value*.*
- *The set* $Val(L)$ *of values is defined by* $Val(L) \stackrel{\text{def}}{=} L \cup Atoms \cup \{nil, \perp\}$.
- env: $Var \rightarrow Val(L)$ *is the* environment *mapping variables into their values.*
- car, cdr: $L \rightarrow Val(L)$ *describes the contents of the car and cdr fields, respectively. We extend the definition of car and cdr to operate on* $Val(L)$*, with* $car(v) = cdr(v) = \perp$ *for every* $v \in Atoms \cup \{nil, \perp\}$.

EXAMPLE 2. Fig. 7 shows the store of the running example program (from Fig. 3) just before the loop (on the input beginning with 'A').

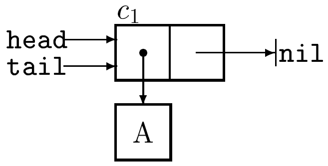
Two-Level Store	Graphical Representation
$\begin{array}{lcl} L & = & \{c_1\} \\ env & = & [\mathtt{h}ead \mapsto c_1, \mathtt{t}ail \mapsto c_1, \mathtt{t}emp \mapsto \perp] \\ car & = & [c_1 \mapsto A] \\ cdr & = & [c_1 \mapsto \mathtt{nil}] \end{array}$	

Fig. 7: The store of the running example program (from Fig. 3) just before the loop (on the input beginning with 'A').

We now use stores to define the meaning of an expression.

DEFINITION 7. (MEANING OF EXPRESSIONS) The meaning of expression e in a store $S = \langle L, env, car, cdr\rangle$ *(denoted by* $[\![e]\!](S)$ *is inductively defined as follows:*

$$\begin{array}{rcl} atom \in Atoms : [\![atom]\!](S) & = & atom \\ [\![\mathtt{nil}]\!](S) & = & \mathtt{nil} \\ v \in Var : [\![v]\!](S) & = & env(v) \\ a \text{ is an access path}, sel \in \{car, cdr\} : [\![a.sel]\!](S) & = & sel([\![a]\!](S)) \end{array}$$

EXAMPLE 3. In the store S shown in Fig. 7, we have:

$$\begin{array}{rcl} [\![\mathtt{head.car}]\!](S) & = & car([\![\mathtt{head}]\!](S)) = car(env(\mathtt{head})) \\ & = & car(c_1) = A \end{array}$$

$$\begin{aligned}[\![\texttt{head.cdr.cdr}]\!](S) &= cdr([\![\texttt{head.cdr}]\!](S)) = cdr(cdr([\![\texttt{head}]\!](S))) \\ &= cdr(cdr(env(\texttt{head}))) = cdr(cdr(c_1)) \\ &= cdr(\texttt{nil}) = \bot\end{aligned}$$

REMARK 1. In Definition 6, $\bot$ is used to avoid the need for partial functions. The advantage of this approach is that the meaning of expressions (Definition 7) is compositional.

Intuitively, the simulation of $\mathcal{P}$ by $ins(\mathcal{P})$ means that for every input I, the sequence of labeled statements executed in $\mathcal{P}$ and $ins(\mathcal{P})$, and the values output, are identical. More formally:

DEFINITION 8. *Let LS be an arbitrary sequence of statement labels in $\mathcal{P}$, and I an input vector. We denote by $I, LS \overset{\mathcal{P}}{\models} S$ the fact that the input I causes LS to be executed in $\mathcal{P}$, and that S is the resultant store.*

We further denote by $I, LS \overset{\mathcal{P}}{\models} e_1 = e_2$, where e_1 and e_2 are expressions, that $I, LS \overset{\mathcal{P}}{\models} S$ and the $[\![e_1]\!](S) = [\![e_2]\!](S)$.

EXAMPLE 4. In the running example, `head.cdr.cdr` and `tail` denote the same cell before the third iteration for inputs of length four or more. Therefore:

$$I, [\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{l}_5]([\mathtt{l}_6, \mathtt{l}_7, \mathtt{l}_8, \mathtt{l}_9, \mathtt{l}_{10}, \mathtt{l}_{11}])^2 \overset{\mathcal{P}}{\models} \texttt{head.cdr.cdr} = \texttt{tail}$$

THEOREM 1. (SIMULATION THEOREM) *Given program $\mathcal{P}$, input I, expressions e_1 and e_2, and sequence of statement labels LS:*

$$I, LS \overset{\mathcal{P}}{\models} e_1 = e_2 \iff I, LS \overset{ins(\mathcal{P})}{\models} ins_R(e_1) = ins_R(e_2)$$

EXAMPLE 5. In the running example, before the first iteration, in the last box of Fig. 6, `head` and `tail` denote the same cell and `head.cdr` and `tail.cdr` denote the same cell. Also, in the instrumented program, `tail.cdr` and `head.cdr.cdr.cdr.cdr` denote the same cell before the third iteration for inputs of length four or more. Finally, as expected from Example 4,

$$I, [\mathtt{l}_1, \mathtt{l}_2, \ldots, \mathtt{l}_5]([\mathtt{l}_6, \mathtt{l}_7, \ldots, \mathtt{l}_{11}])^2 \overset{ins(\mathcal{P})}{\models} ins_R(\texttt{head.cdr.cdr}) = ins_R(\texttt{tail})$$

SKETCH OF PROOF OF THEOREM 1. We begin by showing that the sets of data-, instrumentation-, and statement-cells partition the reachable locations in $ins(\mathcal{P})$, where data-cells are accessible by access-paths of the form $ins_R(a)$, and instrumentation-cells are accessible by access-paths of the form $ins_L(a)$.

Then, we prove by induction on the length of executions, that for the stores $S = \langle L, env, car, cdr \rangle$ (in $\mathcal{P}$) and $S' = \langle L', env', car', cdr' \rangle$ (in $ins(\mathcal{P})$) there exists an injection, d: $Val(L) \to Val(L')$, such that for every expression, e:

$$d([\![e]\!](S)) = [\![ins_R(e)]\!](S') \tag{1}$$

Thus, d is also a bijection between the locations in $\mathcal{P}$ and the data-cells in $ins(\mathcal{P})$. For example, in the stores S and S' shown in last box of Fig. 6, d is defined by:

$$d(v) = \left\{ \begin{array}{cc} c_1' & v = c_1 \\ v & \text{otherwise} \end{array} \right\}$$

Theorem 1 follows directly from Eq. (1).

3.4.2 The main result

We now show that $ins(\mathcal{P})$ also maintains "last wrote" history information for each location (and variable) in $\mathcal{P}$.

LEMMA 1. (LAST WROTE LEMMA) *For every input I, sequence of statement labels LS, label $\mathtt{l_i}$, and access path a, at the end of LS in $ins(\mathcal{P})$ $ins_L(a).\mathtt{car}$ and $\mathtt{pl_i}$ are aliases, i.e.,*

$$I, LS \overset{ins(\mathcal{P})}{\models} ins_L(a).\mathtt{car} = \mathtt{pl_i},$$

if and only if $I, LS \overset{\mathcal{P}}{\models} S$ and one of the following is true: (i) the access path a is a simple variable v that was last assigned in $\mathcal{P}$ at $\mathtt{l_i}$. (ii) the access path a has the form $a'.\mathtt{sel}$ and the sel field of the location $[\![a']\!](S)$ was last assigned in $\mathcal{P}$ at $\mathtt{l_i}$.

EXAMPLE 6. In the running example, before the first iteration, in the last box of Fig. 6, we have

$$I, [\mathtt{l_1}, \mathtt{l_2}, \mathtt{l_3}, \mathtt{l_4}, \mathtt{l_5}] \overset{ins(\mathcal{P})}{\models} ins_L(\mathtt{head}).\mathtt{car} = \mathtt{pl_1}$$

because $\mathtt{l_1}$ is the statement that last wrote into c_1, which is pointed to by `head`. Also, for input list $I =$ ['A','x'], we have:

$$I, [\mathtt{l_1}, \mathtt{l_2}, \mathtt{l_3}, \mathtt{l_4}, \mathtt{l_5}, \mathtt{l_{12}}, \mathtt{l_{13}}] \overset{ins(\mathcal{P})}{\models} ins_L(\mathtt{tail.car}).\mathtt{car} = \mathtt{pl_2}$$

because for such input $\mathtt{l_2}$ last wrote into the car field of c_1, which is pointed to by `tail.car`. This field was updated by the statement $\mathtt{l_2}$: `read(head.car)`.

SKETCH OF PROOF OF LEMMA 1. We only consider the case that the access path has the form: $a = a'.\mathtt{sel}$. We have to show that at the end of LS, in $ins(\mathcal{P})$, $ins_L(a).\mathtt{car}$ and $\mathtt{pl_i}$ are aliases i.e.,

$$I, LS \overset{ins(\mathcal{P})}{\models} ins_L(a).\mathtt{car} = \mathtt{pl_i}$$

if and only if $I, LS \overset{\mathcal{P}}{\models} S$ and the `sel` field of the location $[\![a']\!](S)$ was last assigned in $\mathcal{P}$ at $\mathtt{l_i}$.

Let $l = [\![a']\!](S)$. Then $[\![ins_L(a'.sel).\mathtt{car}]\!](S') = [\![ins_R(a').\mathtt{sel.car}]\!](S') = car'(sel'([\![ins_R(a')]\!](S'))) = car'(sel'(d([\![a]\!](S)))) = car'(sel'(d(l)))$. Therefore, it is sufficient to show that for every input I, sequence of labels LS, label $\mathtt{l_i}$, stores $S = \langle L, env, car, cdr\rangle$, $S' = \langle L', env', car', cdr'\rangle$ such that $I, LS \overset{\mathcal{P}}{\models} S$ and $I, LS \overset{ins(\mathcal{P})}{\models} S'$, d: $Val(L) \mapsto Val(L')$ that satisfies Eq. (1), $l \in L$,

> The `sel` field of l was last written to at $\mathtt{l_i}$ if and only if
> $car'(sel'(d(l))) = env'(\mathtt{pl_i})$

Proof of the if direction: Assume that $car'(sel'(d(l))) = env'(\mathtt{pl_i})$. Since $env'(\mathtt{pl_i})$ is a unique location, it must be that $car'(sel'(d(l)))$ was assigned in the block $\mathtt{l_i}$ in $ins(\mathcal{P})$. Therefore, the `sel` field of l was assigned in $\mathcal{P}$ at $\mathtt{l_i}$. Now assume that this assignment is not the last one, i.e., there exists another assignment to the `sel` field of l at $\mathtt{l_j}$ in $\mathcal{P}$ between $\mathtt{l_i}$ and the last statement in LS. Therefore, in $ins(\mathcal{P})$, the car field of the `sel` field of $d(l)$ is assigned to the statement-cell pointed to by $\mathtt{pl_j}$. Hence, $car'(sel'(d(l))) \neq env'(\mathtt{pl_i})$, a contradiction.

Proof of the only-if direction: Assume that the `sel` field of l was last written at $\mathtt{l_i}$. Obviously, at the store $S'' = \langle \mathrm{L}'', env'', car'', cdr''\rangle$ just after $\mathtt{l_i}$ in $ins(\mathcal{P})$, it must be that $car''(sel''(d(l))) = env''(\mathtt{pl_i})$. Now assume that this is not true at the end of LS. Since the car field of the `sel` field of $d(l)$ is only assigned in $ins(\mathcal{P})$ at statement block $\mathtt{l_k}$ to statement-cell $\mathtt{pl_k}$, we conclude that the `sel` field of l was not last written at $\mathtt{l_i}$, a contradiction.

We are now able to state the main result.

THEOREM 2. (FLOW-DEPENDENCE REDUCTION) *Given a program $\mathcal{P}$ and any two statements $\mathtt{l_i}$: $\mathtt{st_i}$ and $\mathtt{l_j}$: $\mathtt{st_j}$. The statement at $\mathtt{l_j}$ has a flow dependence on the statement at $\mathtt{l_i}$ (in $\mathcal{P}$) if and only if for one of the access paths $a \in readSet(st_j)$, $\mathtt{pl_i}$ is a may-alias of $ins_L(a).\mathtt{car}$ at $\mathtt{l_j}$ (in $ins(\mathcal{P})$).*

EXAMPLE 7. Suppose we wish to find the flow dependences of the statement labeled $\mathtt{l_5}$ in the running example:

$$\mathtt{l_5}\text{: }\mathtt{if\ tail.car} = \text{‘}\mathtt{x}\text{’}\ \mathtt{goto\ l_{12}}$$

First, Table II and Table III are used to determine the read-set of $\mathtt{l_5}$.

$$\begin{array}{ll} \multicolumn{2}{l}{readSet_R(\{\mathtt{tail.car}) \cup readSet_R(\text{‘}\mathtt{x}\text{’})} \\ = & readSet_R(\mathtt{tail.car}) \cup \emptyset \\ = & readSet_R(\mathtt{tail.car}) \\ = & readSet_R(\mathtt{tail}) \cup \{\mathtt{tail.car}\} \\ = & \{\mathtt{tail}\} \cup \{\mathtt{tail.car}\} \\ = & \{\mathtt{tail}, \mathtt{tail.car}\} \end{array}$$

Second, $ins_L(a)$.`car` is calculated for each a in the read-set.

$ins_L(\texttt{tail}).\texttt{car} = \texttt{tail.car}$
$ins_L(\texttt{tail.car}).\texttt{car} = ins_L(\texttt{tail}).\texttt{cdr.car.car} = \texttt{tail.cdr.car.car}$

Third, the may-aliases to `tail.car` and `tail.cdr.car.car` are calculated by any may-aliases algorithm. Finally, we conclude that $\texttt{l}_5$ is flow dependent on the statements associated with the statement-cells that are among the may-aliases found to `tail.car` and `tail.cdr.car.car`.

Sketch of Proof of Theorem 2. The proof follows immediately from the Last Wrote Lemma, Def. 1, and Def. 3.

From a complexity viewpoint our method is not expensive. The time and space for the program transformation are linear in the size of the original program. In applying Theorem 2, the number of times the may-alias algorithm is invoked is also linear in the size of the original program, or more specifically, proportional to the size of the read-sets. It is most likely that the complexity of the may-alias algorithm will be the dominant cost.

4. Plug and play

An important corollary of Theorem 2 is that an arbitrary conservative may-alias algorithm on $ins(\mathcal{P})$ yields a conservative solution to the flow-dependence problem on $\mathcal{P}$. Existing may-alias algorithms, while conservative, yield results that are difficult to compare. It is instructive to consider the flow dependences computed by these algorithms on the running example program.

- The algorithm of [10] yields the may-aliases shown in column 3 of Table VI. Therefore, on this program, this algorithm yields the exact flow dependences, as shown in Fig. 3.
- The more efficient may-alias algorithms of [25, 14, 16], 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, [14, 16] is not able to identify that `tail.cdr` does not point to `head`. Therefore it yields that `head.car` and $\texttt{pl}_7$ are may-aliases at $\texttt{l}_{12}$, and it will conclude that there value of `head.car` read at $\texttt{l}_{12}$ may be written inside of the loop (at statement $\texttt{l}_7$). Thus, it will conclude that $\texttt{l}_{12}$ has a flow dependence on $\texttt{l}_7$.
- The algorithm of [33] finds, in addition to the correct dependences, superfluous flow dependences in the running example. For example, it finds that $\texttt{l}_5$ has a flow dependence on $\texttt{l}_8$. 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:

TABLE VI: Flow-dependence analysis of the running example (see Fig. 5) using a may-alias oracle.

St	Read-Set	May-Aliases
$\mathtt{l}_1$	$\emptyset$	$\emptyset$
$\mathtt{l}_2$	$\{\mathtt{head}\}$	$\{(\mathtt{head.car}, \mathtt{pl}_1)\}$
$\mathtt{l}_3$	$\{\mathtt{head}\}$	$\{(\mathtt{head.car}, \mathtt{pl}_1)\}$
$\mathtt{l}_4$	$\{\mathtt{head}\}$	$\{(\mathtt{head.car}, \mathtt{pl}_1)\}$
$\mathtt{l}_5$	$\{\mathtt{tail}, \mathtt{tail.car}\}$	$\{(\mathtt{tail.car}, \mathtt{pl}_4), (\mathtt{tail.cdr.car.car}, \mathtt{pl}_2),$ $(\mathtt{tail.car}, \mathtt{pl}_{10}), (\mathtt{tail.cdr.car.car}, \mathtt{pl}_7)\}$
$\mathtt{l}_6$	$\emptyset$	$\emptyset$
$\mathtt{l}_7$	$\{\mathtt{temp}\}$	$\{(\mathtt{temp.car}, \mathtt{pl}_6)\}$
$\mathtt{l}_8$	$\{\mathtt{temp}\}$	$\{(\mathtt{temp.car}, \mathtt{pl}_6)\}$
$\mathtt{l}_9$	$\{\mathtt{tail}, \mathtt{temp}\}$	$\{(\mathtt{tail.car}, \mathtt{pl}_4), (\mathtt{tail.car}, \mathtt{pl}_{10}),$ $(\mathtt{temp.car}, \mathtt{pl}_6)\}$
$\mathtt{l}_{10}$	$\{\mathtt{tail}, \mathtt{tail.cdr}\}$	$\{(\mathtt{tail.car}, \mathtt{pl}_4),$ $(\mathtt{tail.car}, \mathtt{pl}_{10}), (\mathtt{tail.cdr.cdr.car}, \mathtt{pl}_9)\}$
$\mathtt{l}_{11}$	$\emptyset$	$\emptyset$
$\mathtt{l}_{12}$	$\{\mathtt{head}, \mathtt{head.car}\}$	$\{(\mathtt{head.car}, \mathtt{pl}_1), (\mathtt{head.cdr.car.car}, \mathtt{pl}_2)\}$
$\mathtt{l}_{13}$	$\{\mathtt{tail}, \mathtt{tail.car}\}$	$\{(\mathtt{tail.car}, \mathtt{pl}_4), (\mathtt{tail.cdr.car.car}, \mathtt{pl}_2),$ $(\mathtt{tail.car}, \mathtt{pl}_{10}), (\mathtt{tail.cdr.car.car}, \mathtt{pl}_7)\}$

  - 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., [22] by increasing k to $2k$.
  - By modifying 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 algorithm from [32] and found exactly all the flow dependences in (a C version of) the running example.

◦ Flow-insensitive pointer analysis algorithms [4, 35, 34] are becoming increasingly popular due to their reduced time and space requirements. In order to use a flow-insensitive algorithm without losing precision due to the use of the temporary variable `instV` in each code block of the instrumented program, we use a different variable, $\mathtt{instV_i}$, for every statement $\mathtt{l_i}$ in the program. The C program that corresponds to the instrumented running example with the above modification (see Fig. 10) was analyzed by Yong using the flow-insensitive algorithm of Yong, Horwitz, and Reps. This algorithm is a version of Andersen's pointer analysis [4] that has been refined to differentiate among fields of structures [37]. The algorithm yields only one false dependence, between $\mathtt{l}_8$ and $\mathtt{l}_{10}$.

An important concern is the overall practical cost of the analysis and its scalability to large programs.

- In [35], an almost linear flow insensitive may-alias algorithm was developed for C programs. This algorithm can be combined with our linear-time reduction to yield an efficient flow-dependence algorithm for C programs. Furthermore, it seems likely that even more precise flow-insensitive may-alias algorithms such as [4] can be engineered to scale for for large programs.
- Scaling flow-sensitive may-alias algorithms for large programs may be more difficult and may require changes in our reduction or the usage of other techniques such as flow-insensitive type analysis (e.g., see [31]). For example, we may need to optimize the number of extra variable names, statements, and allocation sites.

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 (since normally it is not intended to be executed). 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., [6]). In cases where the instrumented program is intended to be executed, it is possible to reduce the memory usage through cons-cell reuse.

5.1 Calculating aliases using flow dependences

It is worthwhile to mention that flow dependences can be also used to find may-alias information and therefore in essence these problems are equivalent. For example, Fig. 8 contains a program fragment that "checks" if two program variables $\mathtt{v}_1$ and $\mathtt{v}_2$ are may aliases at program point $\mathtt{l}_1$. This program fragment preserves the meaning of the original program and $\mathtt{v}_1$ and $\mathtt{v}_2$ are may aliases at $\mathtt{l}_1$ if and only if $\mathtt{l}_3$ has a flow dependence on $\mathtt{l}_2$. In theory, the quality of any flow-dependence algorithm is limited by the quality of the state of the art in may-alias algorithms. However, there is still the potential to develop a practical flow-dependence algorithm that only computes the relevant may-alias queries on demand [30, 21]. This problem is not addressed in this paper.

```
l1:
    if v1 ≠ nil && not atom(v1) then
        l2:v1.cdr := v1.cdr
    fi
    if v2 ≠ nil && not atom(v2) then
        l3: : write(v2.cdr)
    fi
```

Fig. 8: A program fragment such that $\mathtt{v}_1$ and $\mathtt{v}_2$ are may-aliases at $\mathtt{l}_1$ if and only if $\mathtt{l}_3$ has a flow dependence on $\mathtt{l}_2$.

```
typedef struct listelm {struct listelm* next; char cAtom;} ListElm;

int main() {
   ListElm *head, *tail, *temp;
  l1: head = (ListElm*)malloc(sizeof(ListElm));
  l2: head->cAtom = (char)getchar();
  l3: head->next = NULL;
  l4: tail = head;
  l5: while( tail->cAtom != 'x' ) {
     l6: temp = (ListElm*)malloc(sizeof(ListElm));
     l7: temp->cAtom = (char)getchar();
     l8: temp->next = NULL;
     l9: tail->next = temp;
     l10: tail = tail->next;
   }
  l11: printf("The value of head.car is: %c\n", head->cAtom);
  l12: printf("The value of tail.car is: %c\n", tail->cAtom);
   return 0;
}
```

Fig. 9: C version of the running example program from Fig. 3

5.2 Handling C programs

We have extended the translation of Section 3 to C programs. For example, Fig. 9 shows a C implementation of the running example program from Fig. 3, and Fig. 10 shows its instrumented version. The C instrumentation wraps each program type into a structure that contains a pointer to a statement-cell named `inst`, and a pointer to the data type itself, named `data`. Thus, `inst` plays the role of the car field and `data` plays the role of the cdr field. In order to use a flow-insensitive algorithm without losing precision due to the use of the temporary `instV` in each code block of the instrumented program, we use a different variable, $\mathtt{instV}_\mathtt{i}$, for every statement $\mathtt{l}_\mathtt{i}$ in the program.

In moving from our Lisp-like language to C, we have moved to a much richer language. Our instrumentation transformation is limited by the fact that *every* write has to be recorded. This, for example, prevents us from handling library code whose effect is unknown. Also, the instrumentation

```
typedef int LINE;
struct tagI_char {LINE *inst; char data;};
struct tagI_PListElm {LINE *inst; struct tagI_ListElm *data;};
struct tagI_ListElm {struct tagI_char *cAtom; struct tagI_PListElm *next;};
typedef struct tagI_char I_char;
typedef struct tagI_ListElm I_ListElm;
typedef struct tagI_PListElm I_PListElm;

LINE ls1,ls2,ls3,ls4,ls5,ls6,ls7,ls8,ls9,ls10,ls11,ls12;

int main() {
  I_PListElm *head, *tail, *temp, *instV1, *instV3, *instV4;
  I_PListElm *instV6, *instV8, *instV9, *instV10;
  I_char *instV2, *instV7;

l1: instV1 = (I_PListElm*)malloc(sizeof(I_PListElm));
  instV1->inst = &ls1;
  instV1->data = (I_ListElm*)malloc(sizeof(I_ListElm));
  head = instV1;

l2: instV2 = (I_char*)malloc(sizeof(I_char));
  instV2->data = (char)getchar();
  instV2->inst = &ls2;
  head->data->cAtom = instV2;

l3: instV3 = (I_PListElm*)malloc(sizeof(I_PListElm));
  instV3->data = NULL;
  instV3->inst = &ls3;
  head->data->next = instV3;

l4: instV4 = (I_PListElm*)malloc(sizeof(I_PListElm));
  instV4->inst = &ls4;
  instV4->data = head->data;
  tail = instV4;

l5: while( tail->data->cAtom->data != 'x' ) {

  l6: instV6 = (I_PListElm*)malloc(sizeof(I_PListElm));
    instV6->inst = &ls6;
    instV6->data = (I_ListElm*)malloc(sizeof(I_ListElm));
    temp = instV6;

  l7: instV7 = (I_char*)malloc(sizeof(I_char));
    instV7->data = (char)getchar();
    instV7->inst = &ls7;
    temp->data->cAtom = instV7;

  l8: instV8 = (I_PListElm*)malloc(sizeof(I_PListElm));
    instV8->data = NULL;
    instV8->inst = &ls8;
    temp->data->next = instV8;

  l9: instV9 = (I_PListElm*)malloc(sizeof(I_PListElm));
    instV9->inst = &ls9;
    instV9->data = temp->data;
    tail->data->next = instV9;

  l10: instV10 = (I_PListElm*)malloc(sizeof(I_PListElm));
    instV10->inst = &ls10;
    instV10->data = tail->data->next->data;
    tail = instV10;
  }
l11: printf("The value of head.car is: %c\n", head->data->cAtom->data);
l12: printf("The value of tail.car is: %c\n", tail->data->cAtom->data);
  return 0; }
```

Fig. 10: C version of the instrumented running example program

code changes actual memory locations while only preserving aliases. A C program can cast a pointer into an integer and then check its actual value. Of course, in such cases the instrumented program is not necessarily equivalent to the original program. Furthermore, most existing may-alias algorithms cannot handle such programs.

Acknowledgements

We thank Thomas Ball, Michael Benedikt, Thomas Reps, and Reinhard Wilhelm for their comments that led to substantial improvements in this paper. We thank Martin Alt and Florian Martin for PAG, and their PAG implementation of the algorithm [32] for a C subset. We also thank Rakesh Ghiya, Michael Hind and Suan Hsi Yong for applying their may-alias analysis algorithms to (a C version of) our example program.

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*, White Plains, New York, June 1990, 246–256.

[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*, Volume 983 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995, 33–50.

[4] L. O. Andersen *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] 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*. New York, NY, 1979, 29–41.

[6] D.R. Chase, M. Wegman, and F. Zadeck Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*. New York, NY, 1990, 296–310.

[7] 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*. New York, NY, 1993, 232–245.

[8] K.D. Cooper and K. Kennedy Interprocedural side-effect analysis in linear time. In *SIGPLAN Conference on Programming Languages Design and Implementation*. New York, NY, 1988, 57–66.

[9] 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.*, Washington, DC, 1992, 2–13.

[10] A. Deutsch Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*. New York, NY, 1994, 230–241.

[11] 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*. New York, NY, June 1995, 226–228.

[12] 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.

[13] 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, 1987, 319–349.

[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*. Columbus, Ohio, August 1995. Volume 1033 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995, 515–534.

[15] 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.

[16] R. Ghiya and L.J. Hendren Putting pointer analysis to work. In *ACM Symposium on Principles of Programming Languages*. New York, January 1998.

[17] L. Hendren *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, Ithaca, N.Y., Jan 1990.

[18] L. Hendren and A. Nicolau Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (Jan.), 1990, 35–47.

[19] M. Hind and A. Pioli Assessing the effects of flow-sensitivity on pointer alias analyses. In *5th International Static Analysis Symposium*. Pisa, Italy, September 1998. Springer-Verlag.

[20] S. Horwitz, P. Pfeiffer, and T. Reps Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*. Portland, Oregon, June 1989. Volume 24 of *ACM SIGPLAN Notices*, 28–40.

[21] S. Horwitz, T. Reps, and M. Sagiv Demand interprocedural dataflow analysis. Report TR-1283, University of Wisconsin, Computer Sciences Department, August 1995.

[22] 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*. Prentice-Hall, Englewood Cliffs, NJ, 1981, Chapter 4, 102–131.

[23] 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*. New York, NY, 1982, 66–74.

[24] 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*. New York, NY, 1981, 207–218.

[25] W. Landi and B.G. Ryder Pointer induced aliasing: A problem classification. In *ACM Symposium on Principles of Programming Languages*. New York, NY, January 1991, 93–103.

[26] W. Landi, 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*. 1993, 56–67.

[27] J.R. Larus Refining and classifying data dependences. Unpublished extended abstract, Berkeley, CA, November 1988.

[28] J.R. Larus and P.N. Hilfinger Detecting conflicts between structure accesses. In *SIGPLAN Conference on Programming Languages Design and Implementation*. New York, NY, 1988, 21–34.

[29] 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*. New York, NY, 1984, 177–184.

[30] T. Reps Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*. Edinburgh, Scotland, April 1994. Volume 786 of *Lecture Notes in Computer Science*, Fritzson, P., Editor, Springer-Verlag, 1994, 389–403.

[31] Eric Ruf Partitioning dataflow analysis using types. In *ACM Symposium on Principles of Programming Languages*, New York, NY, 1997, 15–26.
[32] 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.
[33] M. Sagiv, T. Reps, and R. Wilhelm Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems 20*, 1 (Jan.), 1998, 1–50.
[34] M. Shapiro and S. Horwitz Fast and accurate flow-insensitive points-to analysis. In *ACM Symposium on Principles of Programming Languages*. 1997.
[35] B. Steengaard Points-to analysis in linear time. In *ACM Symposium on Principles of Programming Languages*. New York, January 1996.
[36] R.P. Wilson and M.S. Lam Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Languages Design and Implementation*. La Jolla, CA, June 18-21, 1995, 1–12.
[37] S.H. Yong, S. Horwitz, and T. Reps Pointer analysis for programs with structures and casting. Submitted for conference publication, October 1998.