Searching for Points-To Analysis

Glenn Bruns and Satish Chandra, *Member*, *IEEE*

Abstract—The points-to analysis problem is to find the pointer relationships that could arise during program execution. Many points-to analysis algorithms exist, each making a particular trade off between cost of the analysis and precision of the results. In this paper, we show how points-to analysis algorithms can be defined as transformed versions of an exact algorithm. We present a set of program transformations over a general program model and use them to define some existing points-to analysis algorithms. Doing so makes explicit the approximations involved in these algorithms. We also show how the transformations can be used to define new points-to analysis algorithms. Our transformations are generic and may be useful in the design of other program analysis algorithms.

Index Terms—Points-to analysis, program analysis, reachability analysis, model checking.

1 Introduction

Points-to analysis is an established topic in program analysis [16]. Algorithms that carry out points-to analysis efficiently on large programs have been developed [13], [7], and some commercial compilers incorporate optimizations that rely on points-to analysis [11]. Also, points-to analyses have been formalized in the frameworks of iterative data-flow analysis [21], set constraints [10], graph reachability [27], and logic programming [29].

Despite all this, one could ask for something more. It is well known that efficient algorithms for points-to analysis give approximate results, but less well known is the degree or nature of the approximation relative to exact results that could be computed although at high cost. It is rare to see these approximations spelled out and their properties studied. Often, existing algorithms apply a series of independent approximations all at once, even though it is possible to use them in isolation. Thus, it is hard to compare algorithms and understand the significance of the analysis results.

In this paper, we seek a detailed understanding of how efficiency is achieved in points-to analysis or, put another way, we seek an understanding of the approximations that are made in points-to analysis. Our approach is to first define points-to analysis as a reachability analysis problem over a labeled transition system (LTS). Using reachability analysis directly would, in practice, be unnecessarily expensive for points-to analysis. However, the framework is still useful because, within it, we can understand how efficiency is achieved in common points-to analysis algorithms. Next, we define a set of program transformations on a general program model. For each transformation, we show the approximation that is introduced with respect to reachability properties. We then show how the transformations can be applied to define some of the existing points-to analysis algorithms, such as Andersen's [1] and Hind et al.'s [15]. In effect, we show how the efficient points-to analysis of a program can be described as reachability analysis on a transformed version of the program. Our main contributions are the following:

1. We define five program transformations, combinations of which are used implicitly in several popular points-to analysis algorithms. With one exception, these transformations are *generic*, in the sense that they do not pertain specifically to points-to analysis.
2. We clarify the relationships between some of the existing points-to analysis algorithms by refactoring them as various combinations of generic program transformations.

In the following section, we present a high-level sketch of how efficient points-to analysis algorithms can be viewed as approximate reachability analysis algorithms. In Section 3, we define the points-to analysis problem. In Section 4, we define a program model; in Section 5, we show how pointer programs are captured in the model; and, in Section 6, we define reachability properties for the model. In the core of the paper, Section 7, we define five program transformations: flow-insensitivity, state accumulation, state merging, local accumulation, and data-flow approximation. In Section 8, we briefly describe three existing algorithms for points-to analysis and show how these can be seen as reachability analyses of transformed programs. Section 9 briefly summarizes and discusses whether our program transformations can be viewed as abstract interpretations.

2 Overview

The points-to analysis problem is to find the pointer relationships that could arise during program execution. Consider a points-to analysis of the simple program in Fig. 1. An analysis of this program with Andersen's algorithm [1] works by incrementally building up a set of points-to facts. Throughout this paper, we use the notation $p \rightarrow q$ to denote that the variable p points to, i.e., contains the address of, the variable q. We begin with the empty set and then, in one step, add the effect of all statements to obtain $\{s \rightarrow q, x \rightarrow y, s \rightarrow p, x \rightarrow z\}$ and take another step to

- *G. Bruns is with Bell Laboratories, Lucent Technologies, 2701 Lucent Lane, Lisle, IL 60532. E-mail: grb@research.bell-labs.com.*
- *S. Chandra is with IBM India Research Laboratory, I.I.T. Hauz Khas, New Delhi 110016 India. E-mail: satishchandra@in.ibm.com.*

Manuscript received 9 Mar. 2003; revised 5 June 2003; accepted 21 June 2003. Recommended for acceptance by W. Griswold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118734.

```
if (...) {
      s = &q;
      x = &y;
}
else {
      s = &p;
      x = &z;
}
*s = x;
```

Fig. 1. A C program involving pointers.

obtain $\{s \to q, x \to y, s \to p, x \to z, p \to y, p \to z, q \to y, q \to z\}$. Taking another step would not enlarge the set of points-to facts, so the algorithm terminates.

Fig. 2 (left) depicts the analysis graphically. The nodes of the figure represent the sets of points-to facts at successive steps of analysis. The final set is only an approximation of the points-to relationships that can actually occur during execution of the program. For example, it is not possible for p to point to y during any program execution, but $p \to y$ is shown in the final set. On the other hand, if a variable can point to another variable during program execution, this fact is in the final set.

Now, consider an exact points-to analysis of this program, shown on the right of Fig. 2. Each node represents the set of points-to facts that hold at a particular point in the program's execution. Each edge represents the execution of a program statement and is labeled with the statement. Asking whether p can point to y during program execution amounts to asking whether a node containing $p \to y$ can be reached from the node representing the initial program state (ignoring the outcome of conditionals).

The two graphs in Fig. 2 are similar. Both the nodes contain sets of points-to facts, and the edges represent the effect of program statements. This observation leads us to ask whether Andersen's analysis can be regarded as a kind of approximate reachability analysis. In particular, can the program be transformed so that reachability analysis of the transformed program gives us Andersen's analysis?

In answering this question, it is helpful to consider the differences between the graphs of Fig. 2. In the left-hand graph, there is no branching, the statement order in the program is ignored, each node has an outgoing edge, and no points-to fact is ever removed through an edge. In the right-hand graph, branching does occur, statement order matters, and some nodes have no outgoing edges. Although not illustrated by the right-hand graph, in an exact analysis, a points-to fact at a node can be deleted through an edge. The program transformations described later in the paper are related to these features. For example, we define program transformations that have the effect of eliminating branching in the program's graph, ensuring that a program state can only "grow" through a program transition, and ignoring program statement order.

In this paper, we study how various points-to algorithms can be framed as approximate reachability problems. One benefit of this work is in making explicit the approximations behind the algorithms. But, more importantly, the work suggests how one might derive approximate solutions to other program analysis problems that can be defined as reachability problems.

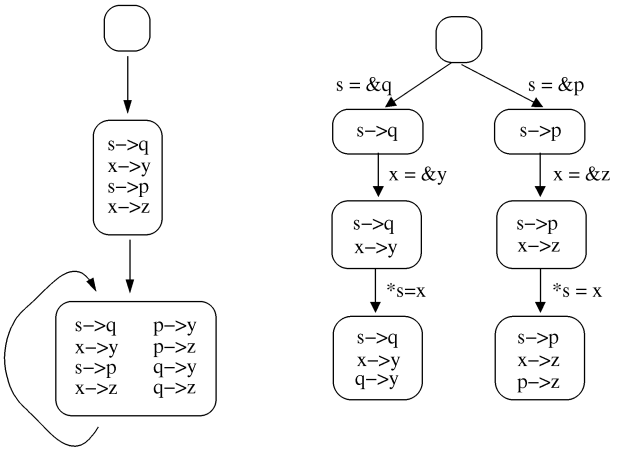


Fig. 2. Andersen's analysis (left) and an exact analysis (right).

3 Points-To Analysis

We define the points-to analysis problem as a variant of the *intraprocedural*, *may*, *flow sensitive*, pointer aliasing problem defined by Landi and Ryder [22].

A *control-flow graph* (CFG) is a graph representation of a program in which nodes are program statements and edges represent control flow. Fig. 3 shows the CFG of the program in Fig. 1. Every CFG contains distinguished `Entry` and `Exit` statements. A conditional statement is modeled as nondeterministic control flow from the preceding statement to the statements of the branches.

The key observation about CFGs is that control and data are independent. Control flow is defined by the edges of the CFG and data update by the statements at the nodes.

A *pointer program* is a CFG in which statements come in only four forms: `p = &q`, `p = q`, `p = *q`, and `*p = q`. These statements borrow syntax from the C language. The `&` operator takes the address of a variable and the `*` operator dereferences a pointer. Nonpointer statements are not used, and pointer statements with multiple `*`s in front of variable names are rewritten to these four cases by introducing temporary variables. This reduction is not precise for the flow-insensitive case [17].

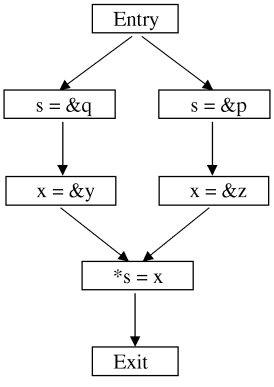


Fig. 3. An example control-flow graph.

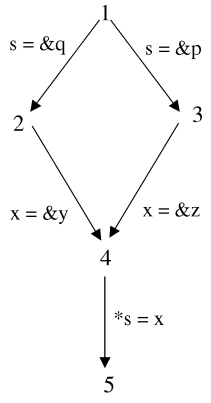


Fig. 4. An alternative representation of the CFG in Fig. 3.

Definition 1. *Given a CFG, the* precise points-to problem *is to determine whether a given variable p points to another variable q after executing some path from* `Entry` *to a given node n.*

We have omitted two important features of real-world programs from our problem definition. First, we do not have procedure call and return statements in our CFGs. Rather, we fit a program into our restricted CFGs by approximating calls and returns by gotos, without regard to valid interprocedural execution [30]. Many papers, e.g., [21], [9], have presented ways to avoid this imprecision, albeit approximately. Our second omission is that we do not have *malloc* statements in our CFGs. In fact, adding dynamic memory allocation to our program representation would make the precise may-alias problem undecidable ([3], Theorem 2). Rather, we replace each occurrence of *malloc* by a static address to fit our restricted CFGs. Several papers, e.g., [21], [8], have presented less drastic, yet computable approximations of dynamic memory allocation.

Even under the restrictions described above, the precise points-to analysis problem is NP-hard for CFGs with multilevel pointers [22], [20]. By multilevel pointers, we mean that pointers can contain addresses of pointer variables. Practical polynomial-time pointer analysis algorithms for C-like languages compute only approximate answers to the points-to problem.

4 Program Model

We now define a simple, abstract program model. We say "program model" and not "programming language" because, for simplicity, we say nothing about syntax but instead work directly at the semantic level of sets and functions.

Assume as given a set $State$ of states. A *program* over $State$ is a set of statements, each having the form $(en, next)$, where $en : State \to Bool$ is an "enablement" predicate and $next : State \to State$ is a "next state" or "state update" function. Intuitively, if predicate en holds at the current program state, then $next$ can be applied to obtain a new state. We write $\mathrm{Prog}(State)$ for the set of programs over $State$.

The meaning of a program is defined as a labeled transition system (LTS), which is a directed graph in which a node represents a program state and an edge represents the effect of a program statement.

TABLE 1
The CFG Program for the CFG in Fig. 3

$en(c)$	$cnext(c)$	$dnext(d)$
$c = 1$	2	$[\mathtt{s} = \&\mathtt{q}](d)$
$c = 1$	3	$[\mathtt{s} = \&\mathtt{p}](d)$
$c = 2$	4	$[\mathtt{x} = \&\mathtt{y}](d)$
$c = 3$	4	$[\mathtt{x} = \&\mathtt{z}](d)$
$c = 4$	5	$[*\mathtt{s} = \mathtt{x}](d)$

Definition 2. *Let P be a program over $State$. The LTS for P over $State$, written $(P, State)$, has $State$ as its node set and a transition relation defined as follows:*

$$s \to s' \triangleq \exists (en, next) \in P \text{ s.t. } en(s) \text{ and } s' = next(s).$$

We write $s \to^* s'$ if there exists a path from s to s' within an LTS.

Some of the program transformations we define require that control and data are independent, as in CFGs. We therefore define a class of programs with this property. A *Control-Data program* (*CD program* for short) over $Control \times Data$ is a program in which each statement can be written in the form $(en, (cnext, dnext))$, where $en : Control \to Bool$ is an enablement predicate, $cnext : Control \to Control$ is a "control update" function, and $dnext : Data \to Data$ is a "data update" function. We write $\mathrm{CD}(Control, Data)$ for the set of CD programs over $Control \times Data$.

Similarly, some of our program transformations require a simple and fixed control structure. In particular, the requirement is that the enablement predicate compares the current control value to a constant and that the control update function maps the current control value to a constant. Programs obtained by translation from CFGs have this property. We write at_c for the enablement predicate $\lambda c'.(c = c')$ and to_c for the control update function $\lambda c'.c$. We write $\mathrm{CFG}(Control, Data)$ for the set of CD programs over $Control \times Data$ for which every statement can be written in the form $(at_c, (to_{c'}, dnext))$, and call these programs *CFG programs*.

We now describe how a CFG can be translated to a CFG program. For this purpose, it is convenient to use an alternative graph representation of the CFG in which nodes represent control points and edges are labeled with statements. Fig. 4 shows the CFG of Fig. 3 in this representation. From such a graph, it is simple to derive a program. Let $Control$ be the set of node labels in the graph and assume that semantic function $[\_]$ maps statements in the graph to functions from $Data$ to $Data$. Then, the program in $\mathrm{CFG}(Control, Data)$ representing the graph has a statement $(at_m, (to_n, [stmt]))$ for every edge from m to n labeled with $stmt$ in the graph. In later sections, we always present CFGs in the style of Fig. 4.

Translating the CFG in Fig. 3 gives the CFG program of Table 1, where each line describes a program statement of the form $(en, (cnext, dnext))$. Note that *en* and *cnext* have the special form of at_c and to_c, respectively. The first statement can be read as "if at control point 1, then apply the data

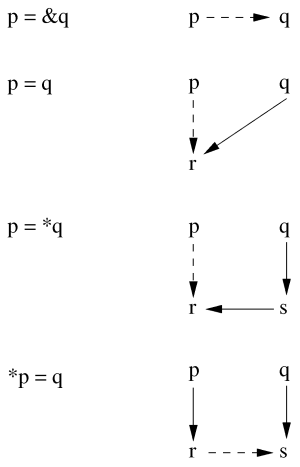


Fig. 5. An informal presentation of pointer statement semantics.

update meant by statement `s = &q` and move to new control point 2."

The program transformations we define in later sections are sometimes defined on basic programs, sometimes on CD programs, and sometimes on CFG programs. Clearly, a transformation defined on basic programs can be applied to CD and CFG programs. For example, suppose we have a CD program P in $\mathrm{CD}(Control, Data)$. We can derive a program P' in $\mathrm{Prog}(Control \times Data)$ by deriving from each statement $(en, (cnext, dnext))$ of P a statement $(en', next)$ of P', where $en'(c, d) = en(c)$ and $next(c, d) = (cnext(c), dnext(d))$. A transformation defined on CD programs can be directly applied to CFG programs.

5 Semantics of Pointer Statements

Our translation of CFGs to programs is defined relative to a semantics that maps statements to data update functions. In this section, we study the semantics of pointer statements. We first present a basic semantics for pointer statements in which variable bindings are represented as a variable-to-variable mapping. However, points-to analyses are usually based on a semantics in which variable bindings are represented by a binary relation on variables. We therefore consider how one might derive such a relation semantics for pointer statements from the basic semantics. We show a general method for performing the derivation and argue that it is natural because, in a specific formal sense, it gives the most precise relation semantics that is safe relative to the basic semantics. Finally, we show that our derived relation semantics for pointer statements is more precise than a relation semantics commonly used in the points-to analysis literature. The material in this section is worked out in detail because it is useful to see how a pointer statement semantics itself can be a hidden source of approximation in points-to analysis.

5.1 Basic Semantics

In denotational semantics, it is common to model variable bindings as an "environment" or "store" that maps variables to their values. Given a set Var of variables and a set Val of values, we define $Env = Var \to Val_\perp$ as the set of environments, where $Val_\perp$ is the set Val extended with the distinguished constant $\perp$. In the case of pointer statements, the set Val of values is Var. We now define the meaning $[stmt]$ of each pointer statement as a mapping from environments to environments. When e is an environment, we write $e[x := y]$ for the environment that is like e except that x is mapped to y and $e_\perp$ for the environment that maps all variables to $\perp$.

$$
\begin{array}{rcl}
[\mathtt{p = \&q}](e) & = & e[\mathtt{p} := \mathtt{q}] \\
[\mathtt{p = q}](e) & = & e[\mathtt{p} := \mathtt{e(q)}] \\
[\mathtt{p = *q}](e) & = & \text{if } e(\mathtt{q}) = \perp \text{ then } e[\mathtt{p} := \perp] \text{else } e[\mathtt{p} := e(e(\mathtt{q}))] \\
[\mathtt{*p = q}](e) & = & \text{if } e(\mathtt{p}) = \perp \text{ then } e_\perp \text{ else } e[e(\mathtt{p}) := e(\mathtt{q})]
\end{array}
$$

Fig. 5 provides some intuition about the pointer statements. For each statement, there is a graph showing how the statement modifies a particular environment. The solid arrows represent the environment prior to statement execution; the dotted and solid arrows represent the environment after statement execution. If we define an order $\leq$ on environments by $e_1 \leq e_2$ iff for all variables x, $e_1(x) = \perp$ or $e_1(x) = e_2(x)$, then each pointer statement is monotonic.

Using the semantics above, a state of a pointer program has the form (c, e), where c is a control point and e is an environment. Fig. 6 shows an LTS for the CFG of Fig. 3. Each state is shown as a rounded rectangle, with the control point on the left and the environment on the right. For each program variable x, we write $x \to y$ if the environment maps x to y and write nothing if the environment maps x to $\perp$. Although a CFG and its corresponding LTS may seem to have similar shapes, they are quite different. Each edge in the CFG represents a unique control point, while multiple nodes in the LTS can correspond to a single control point. For example, in the LTS of Fig. 6, two states are associated with control point 6.

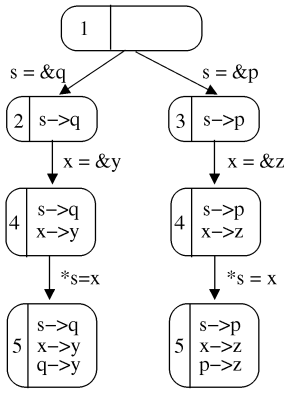


Fig. 6. The labeled transition system for the CFG of Fig. 3.

5.2 Relation Semantics

Points-to analysis algorithms typically model pointer relationships not as environments, but as relations, as in Fig. 2. One motivation for doing so is to support a natural sense of "merging" or "joining" information about pointer variables.

Given a set Var of variables and a set Val of values, we define $Rel = \mathrm{Pow}(Var \times Val)$ as the set of relations over Var

and Val. We can derive a semantics for statements over relations from a semantics for statements over environments. Informally, the relation semantics is as follows: Given a relation R, consider each environment e that is compatible with R, apply the basic semantics of the statement to each e, and then combine the resulting environments. The function γ maps a relation R to the set of compatible environments; the function α forms a relation from a set E of environments:

$$\gamma(R) = \{e \mid \forall x \in Var : e(x) \in Val \Rightarrow (x, e(x)) \in R\},$$
$$\alpha(E) = \{(x, e(x)) \mid e \in E \text{ and } x \in Var \text{ and } e(x) \in Val\}.$$

Then, if $[stmt] : Env \to Env$ is the basic semantics for statement $stmt$, the derived relation semantics for $stmt$ is $[stmt]_{rel} : Rel \to Rel$, defined as follows:

$$[stmt]_{rel}(R) = \alpha\{[stmt](e) \mid e \in \gamma(R)\}.$$

Other relation semantics are possible, but, by the theory of abstract interpretation [6], [18], this semantics is the most precise semantics that is safe with respect to the basic semantics. To briefly justify this claim, we note that functions α and γ form a Galois insertion, where an element of Rel is regarded as an abstract representation of a set of elements of Env. The relation semantics $[\_]_{rel}$ is a *safe* approximation of basic semantics $[\_]$ if it satisfies the condition that $\{[stmt](e) \mid e \text{ in } E\} \subseteq \gamma([stmt]_{rel}(\alpha(E)))$ for every set E of environments. Lemma 2 of [18] tells us that not only is $[\_]_{rel}$ safe, but it is the most precise of all safe relation semantics. In other words, applying any other safe relation semantics to a relation R will yield a relation larger in subset ordering than what one would obtain by applying $[\_]_{rel}$.

Note that $[stmt]_{rel}$ is monotonic with respect to the subset inclusion ordering on Rel. This fact does not depend on whether $[stmt]$ is monotonic.

Definition 3. *Suppose* $P \in \mathrm{CD}(Control, Var \to Val_\perp)$. *Then, the* relation-based version $\mathrm{Rel}(P)$ *of* P *is a program in* $\mathrm{CD}(Control, \mathrm{Pow}(Var \times Val))$ *that contains, for each statement* $(en, (cnext, dnext))$ *of* P, *a statement* $(en, (cnext, dnext_{rel}))$.

If a state is reachable in P, then a corresponding state is reachable in $\mathrm{Rel}(P)$. To make this precise, we use the functions α and γ defined earlier in this section. We say that a state (c, e) of a program P and a state (c, R) of $\mathrm{Rel}(P)$ *correspond* if $\alpha(\{e\}) \subseteq R$.

Proposition 1. *Suppose* $P \in \mathrm{CD}(Control, Var \to Val_\perp)$. *If state* s *of* P *corresponds to a state* t *of* $\mathrm{Rel}(P)$ *and* s' *is reachable from* s, *then a state* t' *corresponding to* s' *is reachable from* t.

Thus, $\mathrm{Rel}(P)$ is an approximation of program P. For points-to analysis, this means that an analysis using relation semantics for pointer statements may give a result that includes pointer relationships that could not actually occur.

5.3 Customary Pointer Statement Semantics

In the previous section, we showed how to derive a relation-based statement semantics from an environment-based semantics. This derivation can be used with any environment-based semantics, but we are interested in using it to derive a relation-based semantics for pointer statements. If $stmt$ is a pointer statement and $[stmt]$ is the interpretation of the statement as a mapping from environments to environments, then $[stmt]_{rel}$ is the interpretation of the statement as a mapping from relations to relations.

A drawback of a relation-based semantics for pointer statements defined in this way is that it is hard to read. For example, suppose we want to understand the relation-based interpretation of statement $[\mathtt{p} = \mathtt{\&q}]$. The definition of the previous section tells us that

$$[\mathtt{p} = \mathtt{\&q}]_{rel}(R) = \alpha(\{e[\mathtt{p} := \mathtt{q}] \mid e \in \gamma(R)\}).$$

It is more convenient to have a semantic definition that works directly on relations. For example, the right side of the definition above can equivalently be written as $(R - \{(\mathtt{p}, r) \mid (\mathtt{p}, r) \in R\}) \cup \{(\mathtt{p}, \mathtt{q})\}$. We now present a direct relation-based semantics for pointer statements.

To simplify presentation of the semantics, we adopt some notation. We write $R \setminus x$ for the relation that is like R except that all pairs of the form $(x, \_)$ are removed. If R_1 and R_2 are relations, then we write $R_1 \sqsubseteq R_2$ to mean that $R_1 \subseteq R_2$ and, if (x, y_1) and (x, y_2) are elements of R_1, then $y_1 = y_2$.

For pointer statements, we refer to the binary relation R on variables that is updated by a statement as a *points-to relation*. In keeping with the notation used so far in this paper, we write an element of a points-to relation as $p \to q$ rather than (p, q). We now define the meaning of each pointer statement $stmt$ as a map $[\![stmt]\!]$ from points-to relations to points-to relations.

$$\begin{aligned}
[\![\mathtt{p} = \mathtt{\&q}]\!](R) &= R \setminus \mathtt{p} \cup \{\mathtt{p} \to \mathtt{q}\} \\
[\![\mathtt{p} = \mathtt{q}]\!](R) &= R \setminus \mathtt{p} \cup \{\mathtt{p} \to r \mid \mathtt{q} \to r \in R\} \\
[\![\mathtt{p} = *\mathtt{q}]\!](R) &= R \setminus \mathtt{p} \cup \{\mathtt{p} \to r \mid \exists s : \{\mathtt{q} \to s, s \to r\} \sqsubseteq R\} \\
[\![*\mathtt{p} = \mathtt{q}]\!](R) &= \{r \to s \mid \{\mathtt{p} \to r, \mathtt{q} \to s\} \sqsubseteq R\} \cup \\
&\quad \{r \to s \mid \{r \to s, \mathtt{p} \to t\} \sqsubseteq R \text{ and } r \neq t\}
\end{aligned}$$

The definition for $\mathtt{p} = \mathtt{\&q}$ says to update the points-to relation R by removing all points-to facts of the form $\mathtt{p} \to \_$ and adding $\mathtt{p} \to \mathtt{q}$.

The definition for $\mathtt{p} = \mathtt{q}$ says to update R by removing all points-to facts of the form $\mathtt{p} \to \_$ and adding $\mathtt{p} \to r$ for every r pointed to by $\mathtt{q}$.

The definition for $\mathtt{p} = {}^*\mathtt{q}$ says to update R by removing all points-to facts of the form $\mathtt{p} \to \_$ and adding $\mathtt{p} \to r$ for all variables r under the following conditions: 1) there exists a variable s such that $\mathtt{q} \to \mathtt{s}$, 2) $s \to r$, and 3) if s happens to be the same as $\mathtt{q}$, then r can only be $\mathtt{q}$. This third requirement is needed to ensure that the direct relation semantics matches the indirect relation semantics of Section 5.2. It will be explained more below.

The definition for ${}^*\mathtt{p} = \mathtt{q}$ is harder to paraphrase. The first clause of the union introduces new points-to facts: If $\mathtt{p}$ points to some r and $\mathtt{q}$ points to some s, then add $r \to s$, except if $\mathtt{p}$ and $\mathtt{q}$ are the same, then r and s must also be the same. The second clause of the union copies existing facts in R, less the ones that must be removed: A fact of the form $t \to \_$ must be removed from R if the only points-to fact in R of the form $\mathtt{p} \to \_$ is $\mathtt{p} \to t$. Note also that if R contains no fact of the form $\mathtt{p} \to \_$, then no points-to facts in R are copied by the second clause.

TABLE 2
The Points-To-Based Version of the Program of Table 1

$en(C)$	$cnext(C)$	$dnext(R)$
$1 \in C$	$\{2\}$	$R \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{q}\}$
$1 \in C$	$\{3\}$	$R \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{p}\}$
$2 \in C$	$\{4\}$	$R \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{y}\}$
$3 \in C$	$\{4\}$	$R \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{z}\}$
$4 \in C$	$\{5\}$	$\{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq R\} \cup \{r \to t \mid \{r \to t, \mathtt{s} \to w\} \subseteq R \text{ and } r \neq w\}$

Proposition 2. *Let $stmt$ be a pointer statement and $R \subseteq Var \times Var$ be a points-to relation. Then,*

$$[\![stmt]\!](R) = [stmt]_{rel}(R).$$

This semantics for pointer statements on relations differs from the semantics commonly found in the points-to analysis literature (e.g., in [15] and [1]). The difference is entirely captured by our use of the $\sqsubseteq$ relation. If the standard subset relation $\subseteq$ is used instead, then one gets the customary semantics.

We prefer our relation semantics to the customary one because, as explained above, our semantics is the most precise relation semantics that is safe with respect to the basic semantics. To see that using $\subseteq$ instead of $\sqsubseteq$ would lose precision, suppose we have points-to relation $R = \{\mathtt{q} \to \mathtt{q}, \mathtt{q} \to \mathtt{r}\}$ and pointer statement $\mathtt{p} = *\mathtt{q}$. Applying $[\![\mathtt{p} = {}^{\star}\mathtt{q}]\!]$ to R yields $R \cup \{\mathtt{p} \to \mathtt{q}\}$. If $\subseteq$ were used instead of $\sqsubseteq$, then applying $[\![\mathtt{p} = {}^{\star}\mathtt{q}]\!]$ to R would yield $R \cup \{\mathtt{p} \to \mathtt{q}, \mathtt{p} \to \mathtt{r}\}$. Notice that the use of $\sqsubseteq$ in the definition of $[\![\mathtt{p} = {}^{\star}\mathtt{q}]\!]$ prevents us from selecting $\mathtt{q}$ for s together with $\mathtt{r}$ for r, which would have (imprecisely) given $\mathtt{p} \to \mathtt{r}$.

However, because we want to relate our work to existing algorithms, we define the *customary* semantic function $\|\_\|$ for pointer statements as the function that is like $[\![\_]\!]$ above except that $\subseteq$ is used in place of $\sqsubseteq$.

Definition 4. *Suppose P in $\mathrm{CD}(Control, Var \to Var_\perp)$ is a pointer program. Then, $\mathrm{Rel}_{Ptr}(P)$ of P is a program in $\mathrm{CD}(Control, \mathrm{Pow}(Var \times Var))$ that contains, for each statement $(en, (cnext, [stmt]))$ of P, a statement $(en, (cnext, \|stmt\|))$.*

Given a CD program P on pointer statements, program $\mathrm{Rel}_{Ptr}(P)$ is a version of P that operates on points-to relations using the customary semantics for pointer statements. In contrast, $\mathrm{Rel}(P)$ is a version of P that operates on points-to relations using the relation semantics for pointer statements. Also, while $\mathrm{Rel}(\_)$ can be applied to any CD program on environments, $\mathrm{Rel}_{Ptr}(\_)$ can be applied only to CD programs on pointer statements.

5.4 Ordering Program States

Some of the program transformations we define assume that an order relation $\leq$ is defined on program states such that $(State, \leq)$ is a lattice. If we have a CD program in $\mathrm{CD}(Control, Data)$ and an order $\leq_c$ exists for control values and $\leq_d$ for data values such that $(Control, \leq_c)$ and $(Data, \leq_d)$ are lattices, then we can take the product of these lattices as the lattice on states. So, if (c_1, d_1) and (c_2, d_2) are states of the program, then $(c_1, d_1) \leq (c_2, d_2)$ iff $c_1 \leq_c c_2$ and $d_1 \leq_d d_2$. The join operation $\vee$ of the product lattice can be derived from the join operations $\vee_c$ and $\vee_d$ of the control and data lattices by $(c_1, d_1) \vee (c_2, d_2) = (c_1 \vee_d c_2, d_1 \vee_d d_2)$.

If we have a CD program in which the data values are points-to relations, then the data values naturally form a lattice under subset ordering. However, if the control values are single control points corresponding to nodes in a CFG, then there is no natural ordering on control values. In this case, sets of control points can be used instead of single control points.

Definition 5. *Suppose $P \in \mathrm{CD}(Control, Data)$. Then, $\mathrm{CSet}(P)$ of P is a program in $\mathrm{CD}(\mathrm{Pow}(Control), Data)$ that contains, for each statement $(en, (cnext, dnext))$ of P, a statement $(en', (cnext', dnext))$, where*

$$\begin{aligned} en'(C) &= \exists c \in C : en(c), (cnext, dnext)), \\ cnext'(C) &= \{cnext(C) \mid c \in C\}. \end{aligned}$$

If P is a CFG program, then $\mathrm{CSet}(P)$ is also a CFG program. A statement $(at_c, (to_{c'}, dnext))$ of P becomes $(at_{\{c\}}, (to_{\{c'\}}, dnext))$ of $\mathrm{CSet}(P)$.

In the following sections, all pointer program examples assume the program has the form $\mathrm{CSet}(\mathrm{Rel}_{Ptr}(P))$, where P is a CFG program. Note that if P is a CFG program, then so is $\mathrm{CSet}(\mathrm{Rel}_{Ptr}(P))$. The control values of $\mathrm{CSet}(\mathrm{Rel}_{Ptr}(P))$ are sets of control points; they form a lattice under subset ordering. The data values of $\mathrm{CSet}(\mathrm{Rel}_{Ptr}(P))$ are points-to relations; they also form a lattice under subset ordering. Furthermore, in each statement $(en, (cnext, dnext))$ of program $\mathrm{CSet}(\mathrm{Rel}(P))$, functions $cnext$ and $dnext$ are monotonic and predicate en is monotonic in the sense that, if $C_1 \subseteq C_2$ and $en(C_1)$, then $en(C_2)$.

Table 1 shows the program P for the CFG in Fig. 3. Table 2 shows the points-to-based version $\mathrm{CSet}(\mathrm{Rel}_{Ptr}(P))$.

6 Program Analysis

In this paper, we consider only reachability properties of programs which express that, from a given state, some other state is reachable that satisfies a certain state predicate. Given an LTS $(P, State)$ and a state s of $State$, we write $s, (P, State) \models \mathrm{EF}\phi$ if a state satisfying state predicate ϕ is reachable from s.

Definition 6. *Let $(P, State)$ be an LTS with s in State. Then $s, (P, State) \models \mathrm{EF}\phi$ holds iff there exists an s' in State such that $s \to^* s'$ and $\phi(s')$.*

For points-to analysis, we write state predicates as propositional logic formulas built up from atomic propositions of the form $x \to y$ and $cp = i$ ("cp" is meant to suggest

TABLE 3
The Flow-Insensitive Version of the Program of Table 2

$en(C)$	$cnext(C)$	$dnext(R)$
$true$	$\{2\}$	$R \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{q}\}$
$true$	$\{3\}$	$R \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{p}\}$
$true$	$\{4\}$	$R \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{y}\}$
$true$	$\{4\}$	$R \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{z}\}$
$true$	$\{5\}$	$\{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq R\} \cup \{r \to t \mid \{r \to t, \mathtt{s} \to w\} \subseteq R \text{ and } r \neq w\}$

"control point"). We define the interpretation of these propositions on states of a pointer program P according to the structure of the program's data and control sets. Consider a proposition $x \to y$, where x and y are elements of a set Var. If P has states of the form (c, e), where $e : Var \to Var_\perp$ is an environment, then $x \to y$ holds at (c, e) if $e(x) = y$. If P has states of the form (c, R), where R is a points-to relation, then $x \to y$ holds at (c, R) if $x \to y \in R$. Next, consider a proposition $cp = i$, where cp is an element of a set $Control$. If P has states of the form (c, d), with c in $Control$, then $cp = i$ holds at (c, d) if $c = i$. If P has states of the form (C, d), with C in $\mathrm{Pow}(Control)$, then $x \to y$ holds at (C, d) if $i \in C$.

For example, suppose P is the program of Table 1. Letting $Control$ be the control points of P and Var be the variables of P, we have $State = Control \times (Var \to Var_\perp)$. Then, $(1, e_\perp), (P, State) \models \mathrm{EF}(\mathtt{x} \to \mathtt{z})$ because there is a path in the LTS for this CFG that leads to a state in which the environment maps x to z.

The precise points-to problem of Section 3 can now be expressed as a reachability problem.

Definition 7. *Suppose P is a pointer program with initial state (c_0, d_0). The precise points-to problem of whether $p \to q$ holds after node n can be written*

$$(c_0, d_0), (P, State) \models \mathrm{EF}(cp = n \wedge p \to q).$$

We emphasize that the role of LTSs in this paper is definitional; there is no need to build an explicit LTS-like structure to perform reachability analysis. For example, the analysis can be done with depth-first search in which storage is required only to represent the states that have already been explored. We will see that even simpler search algorithms can be used for reachability analysis with some classes of programs.

The LTS of a CFG can be regarded as an "interpreted CFG"—it contains all possible executions represented by the CFG. While we have defined points-to analysis as a reachability property of the LTS, an alternative is to define it as a more involved property of the CFG itself. For example, in [32], [28], a CFG-like structure is used in which nodes represent program control points and edges are labeled with program actions. An example action is $\mathrm{isModified}_x$, which signifies that the program represented by the edge modifies variable x. Program properties such as liveness are then be expressed as formulas of a fixed-point temporal logic and can be checked with a model checker directly over the CFG-like structure. The same approach could be used to define and check the points-to analysis problem, but, for exact points-to analysis, the temporal logic formula is complicated, and the size of the formula grows exponentially in the number of program variables. For this reason, we choose to define points-to analysis as reachability on an LTS.

7 Program Transformations

In this section, we define some basic program transformations that can be used to derive existing points-to analysis algorithms. We present control-flow insensitivity (Section 7.1), state accumulation (Section 7.2), state merging (Section 7.3), and local accumulation (Section 7.5) as independent transformations.

We also show how a transformation used in logic programming can be used to increase efficiency (Section 7.4), and that data-flow analysis (Section 7.6) can be defined using local accumulation plus control-flow insensitivity.

We use the CFG of Fig. 3 as a running example throughout these subsections. We also give additional examples where necessary to illustrate points not exhibited by the running example.

7.1 Control Flow Insensitivity

A program analysis is said to be *flow-insensitive* if the order of program statements is ignored or if the result of program analysis does not depend on the order of program statements. We use the term to describe programs for which all statements are always enabled. Thus, in a flow-insensitive program, statements can be executed in any order.

Definition 8. *Let $P = \{(en_1, next_1), \ldots, (en_n, next_n)\}$ be a program. Then, the* flow-insensitive *version* $\mathrm{FI}(P)$ *of P is $\{(en, next_1), \ldots, (en, next_n)\}$, where $en(s) \triangleq true$.*

This transformation can be applied to basic programs and, hence, to CD and CFG programs. Applying it to a CD program yields a CD program, but applying it to a CFG program does not necessarily yield a CFG program.

Table 3 shows the flow-insensitive version of the program of Table 2. It is easy to see that if a reachability property holds of a program P, then it will also hold of $\mathrm{FI}(P)$. If a state is reachable in P via some path, then that path will also be able to be traced in $\mathrm{FI}(P)$. However, making a program flow insensitive loses precision in the sense that a reachability property may hold of $\mathrm{FI}(P)$ but not P.

TABLE 4
The Accumulated, Flow-Insensitive Version of the Program of Table 2

$en(C)$	$cnext(C)$	$dnext(R)$
$true$	$C \cup \{2\}$	$R \cup \{\mathtt{s} \to \mathtt{q}\}$
$true$	$C \cup \{3\}$	$R \cup \{\mathtt{s} \to \mathtt{p}\}$
$true$	$C \cup \{4\}$	$R \cup \{\mathtt{x} \to \mathtt{y}\}$
$true$	$C \cup \{4\}$	$R \cup \{\mathtt{x} \to \mathtt{z}\}$
$true$	$C \cup \{5\}$	$R \cup \{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq R\}$

Proposition 3. *Let P be a program over State, with $s \in State$, and ϕ be a predicate on State. Then,*

$$s, (P, State) \models \mathrm{EF}\phi \Leftrightarrow s, (\mathrm{FI}(P), State) \models \mathrm{EF}\phi.$$

We could also define a weaker notion of flow insensitivity in which all statements are enabled in a state as long as some statement is. In other words, the enablement condition for every statement in the program would be $en(s) \triangleq \bigvee\{en_i(s) \mid i \in \{1, \ldots, n\}\}$.

Transformations like this one have been studied in the model checking literature as *abstraction* techniques [2], [5], [25]. Much of this work focuses on making precise the logical effect of abstractions like flow insensitivity. For example, it is known that, by combining states of an LTS, one obtains an LTS that simulates the original one [5]. Furthermore, if the abstracted LTS satisfies a temporal logic formula of a particular class, then so does the original LTS [2].

7.2 State Accumulation

In performing points-to analysis, we search for a program state in which a particular pointer relationship holds. Along a search path from a state to one of its successors, it may be that the pointer relation at the successor state is larger or smaller than in the source state. We might hope to find a particular pointer relationship more quickly by redefining statements so that the pointer relation always increases when traveling from a state to a successor state.

To formalize this idea, we assume the elements of $State$ form a complete lattice $(State, \leq)$. Actually, it would be enough to assume that $(State, \leq)$ be a partial order with join and a bottom element $\perp$, but, for simplicity, we will assume that $(State, \leq)$ is a complete lattice in all that follows:

Definition 9. *Let $(State, \leq)$ be a complete lattice. Then:*

1. *A state predicate $\phi : State \to Bool$ is* up-closed *if $(s \leq s'$ and $\phi(s))$ implies $\phi(s')$.*
2. *A state function $next : State \to State$ is* monotonic *if $s \leq s'$ implies $next(s) \leq next(s')$.*
3. *A statement $(en, next)$ is monotonic if en is up-closed and $next$ is monotonic.*
4. *A program is monotonic if all its statements are.*
5. *A statement $(en, next)$ is* accumulative *if $s \leq next(s)$ for all s in State such that $en(s)$.*
6. *A program is accumulative if all its statements are.*

Note that an accumulative statement is not necessarily monotonic. For instance, let $State$ be $\{0, 1, 2\}$, ordered by the arithmetic order and let $next(x)$ be: If $x = 0$, then 2, else x. Then, $next$ is accumulative but not monotonic.

Definition 10. *Let $P = \{(en_1, next_1), \ldots, (en_n, next_n)\}$ be a program. Then, the* accumulated *version* $\mathrm{AC}(P)$ *of P is $\{(en_1, next'_1), \ldots, (en_n, next'_n)\}$, where*

$$next'_i(s) \triangleq s \vee next_i(s).$$

State accumulation can be applied to basic programs and, hence, also to CD and CFG programs. Applying it to a CD program yields a CD program. To see this, suppose we have a CD program with statement $(en, (cnext, dnext))$. Applying the transformation tells us that, in the transformed program, we have a corresponding statement $(en, next)$, where $next((c, d)) = (c, d) \vee (cnext, dnext)(c, d)$. By definition of $(cnext, dnext)$ and the way $\vee$ is defined for CD program states (see Section 5.4), we have that the transformed statement can equally be written $(en, (cnext', dnext'))$, where $cnext'(c) = c \vee cnext(c)$ and $dnext'(d) = d \vee dnext(d)$. However, applying state accumulation to a CFG program does not necessarily yield a CFG program.

As mentioned in Section 5.5, in a pointer program over points-to relations, the states form a complete lattice and every program statement is monotonic. Table 4 shows the accumulated, flow-insensitive version of the program of Table 3.

Theorem 1. *Let P be a monotonic program over State with s in State and ϕ be an up-closed predicate on State. Then,*

$$s, (P, State) \models \mathrm{EF}\phi \Rightarrow s, (\mathrm{AC}(P), State) \models \mathrm{EF}\phi.$$

In making a program accumulative, one loses precision. Fig. 7 shows a CFG program for which the LTS of the accumulated version has fewer states than the LTS of the original program.

Flow insensitivity and accumulation are independent transformations. Fig. 8 shows a CFG program (CFG 1) for which accumulating the flow-insensitive version loses precision compared to just the flow-insensitive version. The former produces $x \to y$ (see rightmost diagram in Fig. 8) while the latter does not. By contrast, CFG 2 in Fig. 8 shows a CFG program for which the accumulative version is more precise than the flow-insensitive version. The latter produces $q \to y$ (via execution of `p=&y` followed by `q=p`), while the former does not in any execution.

7.3 State Merging

In our program model, multiple statements can be enabled at a state. In searching for a state in which a particular pointer relationship holds, each transition out of a state must be explored. An alternative approach is to consider all transitions at once by merging the destination states.

Definition 11. *Let $P = \{(en_1, next_1), \ldots, (en_n, next_n)\}$ be a program. Then, the* merged *version* $\mathrm{MG}(P)$ *of P is $\{(en, next)\}$, where*

$$\begin{aligned} en(s) &\triangleq \bigvee\{en_i(s) \mid i \in \{1, \ldots, n\}\}, \\ next(s) &\triangleq \bigvee\{next_i(s) \mid en_i(s) \text{ and } i \in \{1, \ldots, n\}\}. \end{aligned}$$

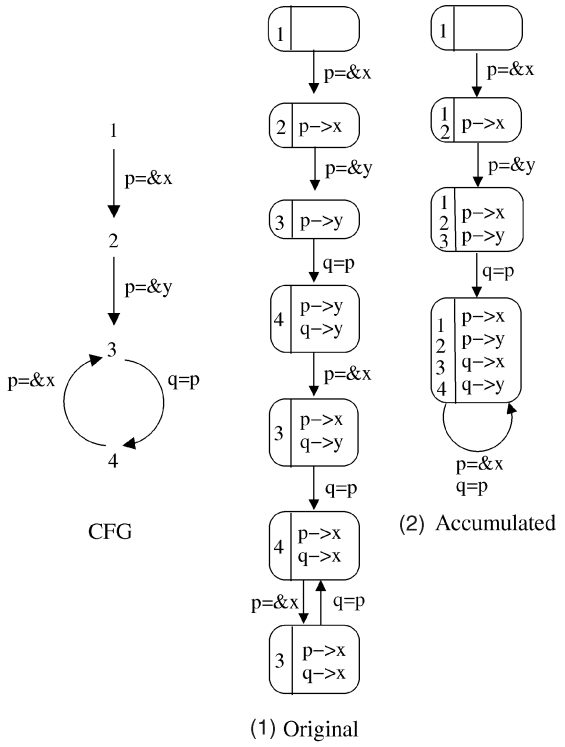


Fig. 7. A CFG for which the LTS of the accumulated version has fewer states.

State merging can be applied to basic programs and, hence, also to CD and CFG programs. Applying it to a CD program yields a CD program, for the same reason that state accumulation does, as explained in the previous section. Applying it to a CFG program does not necessarily yield a CFG program.

The key feature of a merged program is that it has a single state update function. The LTS for a merged program is therefore linear in structure, and reachability analysis involves simply applying the state update function repeatedly until the state of interest is found or until a previously encountered state is met.

Note that merging preserves monotonicity of a program and also preserves accumulativity provided that some program statement is enabled in every element of *State*. Table 5 shows the merged, accumulated, flow-insensitive version of the program of Table 2.

Fig. 9 shows a CFG program for which the LTS of the

1. original version,
2. merged version,
3. accumulative version, and
4. merged, accumulative version

are all different. Merging reduces the number of states in the LTS.

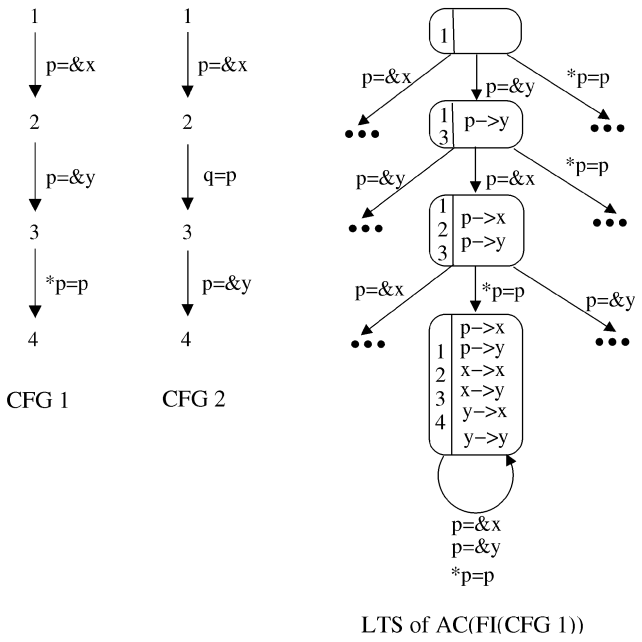


Fig. 8. CFGs to illustrate the independence of the FI and AC transformations.

States that are reachable in a program are also reachable in the merged version of the program.

Theorem 2. *Suppose that P is a monotonic program over $(State, \leq)$, $s \in State$, and ϕ is up-closed. Then,*

$$s, (P, State) \models \mathrm{EF}\phi \Rightarrow s, (\mathrm{MG}(P), State) \models \mathrm{EF}\phi.$$

Generally, states reachable in the merged version of a program P may not be reachable in P itself. However, if P is accumulative and ϕ is up-closed, then P and $\mathrm{MG}(P)$ have exactly the same set of reachable states. The key in showing this is that $(stmt_1; stmt_2)(s) \geq (stmt_1 \vee stmt_2)(s)$ for all states s. Combining this fact with Theorem 2 gives us that merging loses no precision for reachability analysis of monotonic and accumulative programs.

Theorem 3. *Suppose P is a monotonic and accumulative program over $(State, \leq)$, $s \in State$, and ϕ is up-closed. Then,*

$$s, (P, State) \models \mathrm{EF}\phi \;\Leftrightarrow\; s, (\mathrm{MG}(P), State) \models \mathrm{EF}\phi.$$

7.4 Computing State Efficiently

The merged, accumulated, flow-insensitive version of our example program has the form $\{(\lambda s.true, next)\}$, where $next$ is accumulative. Programs of this form are easy to analyze for reachability of an up-closed state predicate: Start with the state of interest and apply $next$ repeatedly until a fixed point is reached (a fixed point is guaranteed if

TABLE 5
The Merged, Accumulated, Flow-Insensitive Version of the Program of Table 2

$en(C)$	$cnext(C)$	$dnext(R)$
true	$C \cup \{2,3,4,5\}$	$R \cup \{\mathtt{s} \to \mathtt{q}, \mathtt{s} \to \mathtt{p}, \mathtt{x} \to \mathtt{y}\} \cup \{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq R\}$

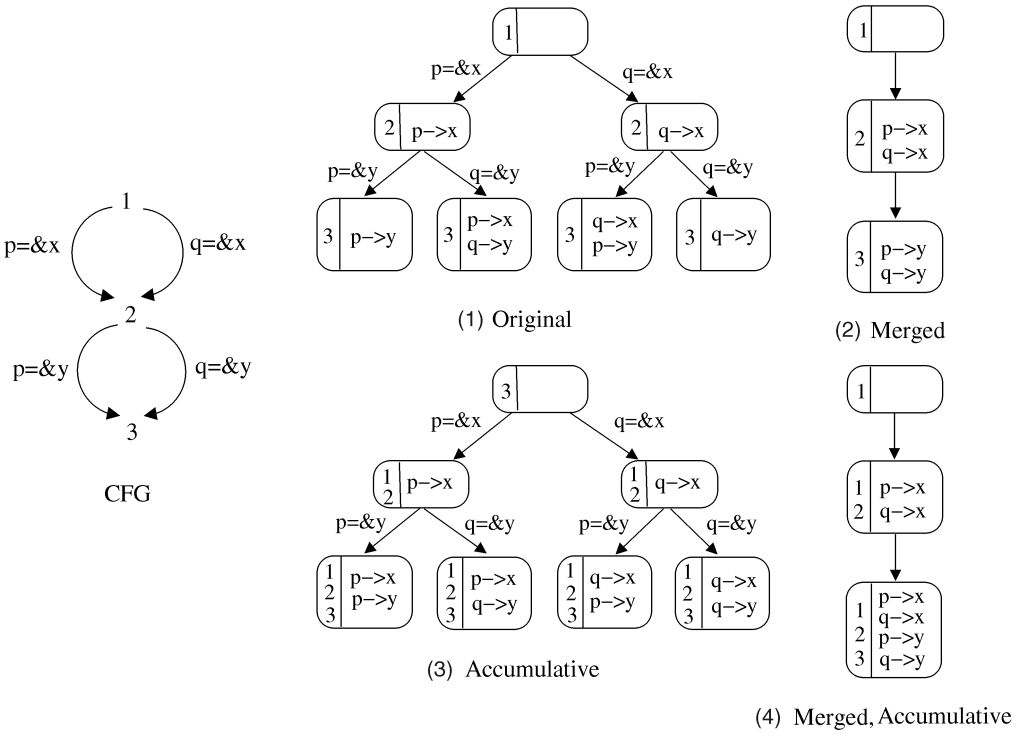


Fig. 9. A CFG to illustrate state merging.

$State$ is a lattice with finite height). Then, check the fixed-point value to see if the predicate holds of it.

The cost of performing this analysis depends not only on the height of the lattice, but also on the cost of applying the function. In this section, we present a program transformation to reduce the cost of the function application. This transformation is unlike the transformations presented in previous sections. First, it is not generic; it can be applied only to accumulative programs having a sole statement that is always enabled. However, many programs can be put in this form, including pointer programs. Second, we do not define a program transformation operation here. We instead show the transformation idea and explain why it reduces the complexity of state updating. Understanding the transformation is important in understanding how cubic complexity is achieved in points-to analysis.

Suppose we are doing reachability analysis of a CFG program that contains pointer statement $\mathtt{x} = {}^{\star}\mathtt{y}$. Intuitively, applying the statement involves adding $\mathtt{x} \rightarrow w$ to points-to relation R if there exist variables z and w such that $\mathtt{y} \rightarrow z$ and $z \rightarrow w$ belong to R. The merged, accumulative, flow-insensitive version of the program will have $dnext$ of the form:

$$dnext(R) = R \cup \{\mathtt{x} \rightarrow w \mid \exists z.\{\mathtt{y} \rightarrow z, z \rightarrow w\} \subseteq R\} \cup \ldots$$

A straightforward implementation of this function would, for the clause of $dnext$ representing $\mathtt{x} = {}^{\star}\mathtt{y}$, iterate over every element $\mathtt{y} \rightarrow z$ of R, then search for an element $z \rightarrow w$ in R. If there were n pointer variables, processing this clause alone could take n^2 steps. Furthermore, in a pointer program, there could be n^2 such clauses because there could be n^2 statements of the form $u = *v$ in the program, giving an overall running time for $dnext$ of $O(n^4)$.

We now transform $dnext$ to $dnext'$, which operates on a state consisting of points-to relation R and a new relation S. We write $x \hookrightarrow y$ for pairs (x, y) when working with S.

$$dnext'((R,S)) =$$
$$(R \cup \{\mathtt{x} \rightarrow w \mid \exists z.\mathtt{x} \hookrightarrow z \in S \text{ and } z \rightarrow w \in R\} \cup \ldots,$$
$$S \cup \{\mathtt{x} \hookrightarrow z \mid \mathtt{y} \rightarrow z \in R\} \cup \ldots).$$

The idea behind the transformation is as follows: Processing the clause shown explicitly on the first line can take n^2 steps as before, but now there can be at most n such clauses, one for each program variable. The worst-case running time for this line is thus $O(n^3)$. Processing the clause on the second line can take at most n steps, but there can be most n^2 such clauses, so the worst-case running time for this line is $O(n^3)$. The overall running time for $dnext'$ is therefore $O(n^3)$.

Correctness of the transformation can be understood in terms of the algorithm sketched above for reachability analysis. The program function is applied repeatedly from a state until a fixed point is reached. This implies that it is enough to show that $dnext$ and $dnext'$ reach the same fixed-point value. They need not update R in the same way along the path to the fixed point.

The complexity results mentioned here can be worked out in detail by writing $dnext$ and $dnext'$ as Datalog programs and using McAllester's Theorem 1 in [24]. The idea of expressing points-to analysis as a logic program is not new. In [29], Shapiro and Horwitz give a Datalog formulation of the points-to analysis problem. In [27], Reps shows how the problem can be formulated as a CFL-reachability problem via a chain-form Datalog program. In [14], Heintze and Tardieu give a demand-driven

TABLE 6
The Locally Accumulative Version of the Program of Table 2

$en(C)$	$cnext(C)$	$dnext(M)$
$1 \in C$	$\{2\}$	$M[2 := M(2) \cup M(1) \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{q}\}]$
$1 \in C$	$\{3\}$	$M[3 := M(3) \cup M(1) \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{p}\}]$
$2 \in C$	$\{4\}$	$M[4 := M(4) \cup M(2) \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{y}\}]$
$3 \in C$	$\{4\}$	$M[4 := M(4) \cup M(3) \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{z}\}]$
$4 \in C$	$\{5\}$	$M[5 := M(5) \cup \{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq M(4)\} \cup \{r \to t \mid \{r \to t, \mathtt{s} \to w\} \subseteq M(4) \text{ and } r \neq w\}]$

algorithm for points-to analysis that uses the same idea to improve on a less efficient demand-driven algorithm. Use of McAllester's theorem improves on earlier work as it clarifies how bottom-up evaluation would give $O(n^4)$ with the original rule and $O(n^3)$ with the revised rules; also, it does not rely on the notion of "chain form" rules.

7.5 Local Accumulation

In the accumulative version of a program, the execution of a program statement leads to a new program state at least as large (according to some order) as the old program state. In terms of a CFG program execution, this means that, along an execution path, the program state will never decrease in passing from one control point to the next.

We can define a weaker notion of accumulation for CFG programs. Along an execution path, the program data value may decrease, but the value computed for a given control point will always be at least as great as the value last computed for the control point *on the same execution path*; the control point may get a smaller value along another execution path in the CFG. Such a program is "locally accumulative" with respect to control points.

Our transformation for local accumulation takes a program in $\mathrm{CFG}(Control, Data)$ and produces a program in $\mathrm{CFG}(Control, Control \to Data)$. Following Schmidt [28], we call a mapping $M : Control \to Data$ that labels control points with data values a *memo table*. We write $M[c := x]$ for the memo table like M except that c is mapped to x. Also, we define the memo table $M_\perp$ by $M_\perp(c) = \perp$ for all c in $Control$. We assume here that $Data$ and $State$ are complete lattices (see Section 7.2).

Definition 12. *Suppose $P \in \mathrm{CFG}(Control, Data)$. The locally accumulative version $\mathrm{LA}(P)$ of P is a program in $\mathrm{CFG}(Control, Control \to Data)$ such that, for every statement $(at_c, (to_{c'}, dnext))$ in P, there is a statement $(at_c, (to_{c'}, dnext'))$ in $\mathrm{LA}(P)$, where*

$$dnext'(M) = M[c' := M(c') \vee dnext(M(c))].$$

Table 6 shows the locally accumulative version of the program of Table 2. No control point is repeated along either of the two program paths, so, in this example, the locally accumulative version computes the same information as the original. Fig. 10 shows a program for which the locally accumulative version is less precise than the original: On path 1-2-3-5-3-4, the former produces extra pairs $q \to z$ and $p \to y$ at control point 4.

Taking the locally accumulative version of a program preserves reachability properties and sometimes introduces no approximation. In the following results, a state predicate ϕ defined on $State = Control \times Data$ is interpreted on a state of the form (c, M) in $State' = Control \times (Control \to Data)$. We define that in such cases $\phi(c, M) = \phi(c, M(c))$.

Proposition 4. *Suppose $P \in \mathrm{CFG}(Control, Data)$, $dnext$ in every statement $(at_{c_1}, (to_{c_2}, dnext))$ of P is monotonic, $(c, d) \in State$, and ϕ is up-closed. Then,*

$$(c, d), (P, State) \models \mathrm{EF}\phi \Rightarrow (c, M_\perp[c := d]), (\mathrm{LA}(P), State') \models \mathrm{EF}\phi.$$

A statement $(en, (cnext, dnext))$ of a CD program is *distributive* if $dnext(a \vee b) = dnext(a) \vee dnext(b)$. A program is distributive if every statement in it is distributive. A predicate ϕ is *existential* if $\phi(x \vee y) \Leftrightarrow \phi(x)$ or $\phi(y)$.

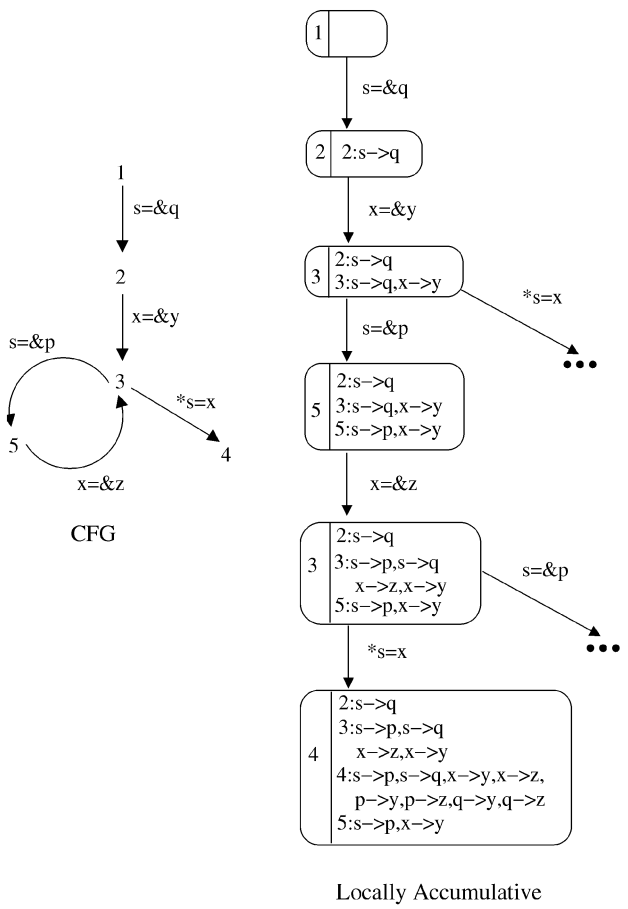


Fig. 10. A CFG to illustrate the effect of local accumulation.

TABLE 7
The Flow-Insensitive, Locally Accumulative Version of the Program of Table 2

$en(C)$	$cnext(C)$	$dnext(M)$
$true$	$\{2\}$	$M[2 := M(2) \cup M(1) \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{q}\}]$
$true$	$\{3\}$	$M[3 := M(3) \cup M(1) \setminus \mathtt{s} \cup \{\mathtt{s} \to \mathtt{p}\}]$
$true$	$\{4\}$	$M[4 := M(4) \cup M(2) \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{y}\}]$
$true$	$\{4\}$	$M[4 := M(4) \cup M(3) \setminus \mathtt{x} \cup \{\mathtt{x} \to \mathtt{y}\}]$
$true$	$\{5\}$	$M[5 := M(5) \cup \{r \to t \mid \{\mathtt{s} \to r, \mathtt{x} \to t\} \subseteq M(4)\} \cup \{r \to t \mid \{r \to t, \mathtt{s} \to w\} \subseteq M(4) \text{ and } r \neq w\}]$

Theorem 4. *In addition to the conditions of Proposition 4, suppose P is distributive and ϕ is existential. Then,*

$$(c,d),(P,State) \models \mathrm{EF}\phi \Rightarrow (c, M_\perp[c := d]),(\mathrm{LA}(P),State') \models \mathrm{EF}\phi.$$

7.6 Data Flow Approximation

Given a program P in $\mathrm{CFG}(Control, Data)$, *data-flow analysis* [6], [26] of P computes the result of the following metaprogram. Its termination is ensured if $Data$ is finite and every data update function $dnext : Data \to Data$ is monotonic.

global $M : Control \to Data$
initially, $\forall i, M(i) = \perp$
repeat until no change to M
  pick a statement $(at_c, (to_{c'}, dnext))$
  $M(c') = M(c') \vee dnext(M(c))$
end

Data-flow analysis can be understood as an efficient but approximate variant of local accumulation. In particular, data-flow analysis is the reachability analysis of the flow-insensitive, locally accumulative version of a program. Note that the use of term "flow-insensitive" here refers to the locally accumulative version of a program and not the original program.

Theorem 5. *Suppose $P \in \mathrm{CFG}(Control, Data)$ with initial state (c,d), P is monotonic, M is the result of data-flow analysis on P, and ϕ is up-closed. Then,*

$$(c, M_\perp[c := d]),(\mathrm{FI}(\mathrm{LA}(P)),State') \models \mathrm{EF}\phi \Leftrightarrow \exists i : \phi(i, M(i)).$$

The efficiency of data-flow analysis comes from the fact that it can visit at most $|Control| \times length(Data)$ states, where $length(Data)$ is the length of longest chain in $Data$. By contrast, the untransformed program may visit up to $|Control| \times |Data|$ states.

Table 7 shows the flow-insensitive, locally accumulative version of the program of Table 2, and Fig. 11 shows a path in the corresponding LTS. Data-flow analysis gives a less precise result than local accumulation: The former produces $q \to z$ and $p \to y$, while the latter does not. (This example is motivated by Landi [19].)

For monotonic CFG programs in which the data-update function of each statement is distributive and for state predicates that are up-closed and existential, data-flow analysis is exact.

Theorem 6. *Suppose $P \in \mathrm{CFG}(Control, Data)$ with initial state (c,d), P is monotonic and distributive, and ϕ is up-closed and existential. Then,*

$$(c,d),(P,State) \models \mathrm{EF}\phi \Leftrightarrow (c, M_\perp[c := d]),(\mathrm{FI}(\mathrm{LA}(P)),State') \models \mathrm{EF}\phi.$$

Data-flow analysis is also related to uninterpreted LTSs. Schmidt [28] shows how modal mu-calculus formulas can be used to express meet-over-all-path (MOP) properties over uninterpreted LTSs. It is well known that, for distributive problems, data-flow analysis computes the same MOP answer [12], [23], but, in general, it may compute a safe approximation.

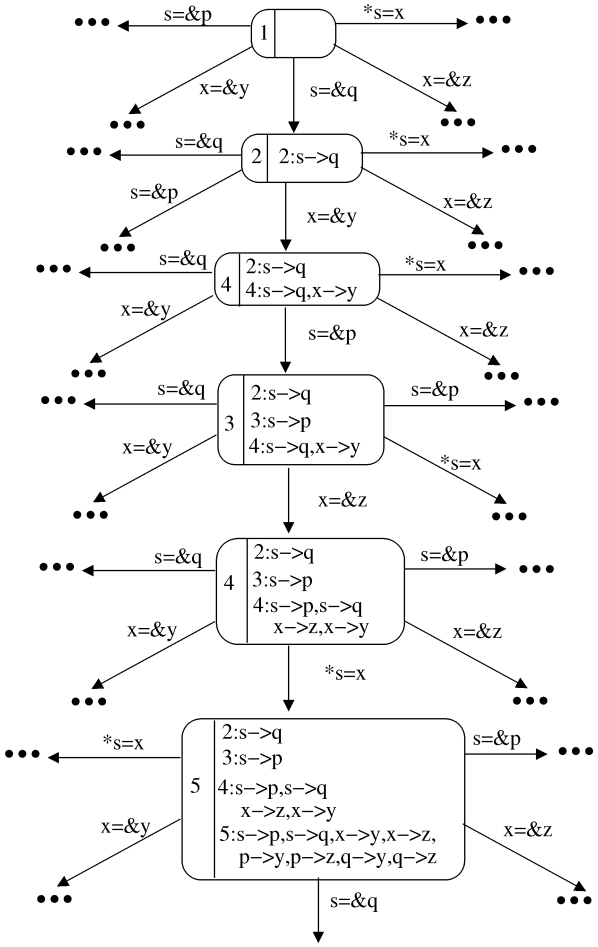


Fig. 11. A path in the LTS for the flow-insensitive, locally accumulative version of the program of Table 2.

8 Points-To Analysis as Reachability Analysis

8.1 Existing Algorithms

We describe briefly the intraprocedural versions of two algorithms for computing safe but approximate answers to the precise points-to problem. Note that we limit our focus to just the core ideas of these algorithms; the actual algorithms handle much more detail than we describe here.

8.1.1 HBCC Algorithms

We sketch the algorithm given in Hind et al. [15]. Our presentation uses points-to relations, while the original uses alias graphs. The algorithm is given by a set of data-flow equations over a CFG in its conventional representation. We assume that each program statement p is associated with In and Out sets, which are collections of points-to facts. The analysis iterates over the following set of equations for each statement, until convergence:

$$\begin{aligned} In[p] &= \bigcup_{q \in pred(p)} Out[q], \\ Out[p] &= (In[p] - MustKill_p(In[p])) \cup Gen_p(In[p]). \end{aligned}$$

The equation for $Out[p]$ essentially computes (p) as defined in Section 5. For each statement, *MustKill* and *Gen* can be gleaned from the structure of (p).

Hind et al. also give a flow-insensitive[1] algorithm in which the control-flow graph is transformed into a large "switch" inside a single loop, and *MustKill* for each statement is ignored.

$$\begin{aligned} In &= \bigcup_{q \in statements} Out[q], \\ Out[p] &= In \cup Gen_p(In). \end{aligned}$$

8.1.2 Andersen's Algorithm

Andersen's algorithm [1] is generally presented as the following set of rules. The idea is to generate facts using the following rules until no more facts can be generated; the final set of facts is the answer for all program points.

$$\begin{aligned} \mathtt{p} = \mathtt{\&q} &: PointsTo(\mathtt{p}, \mathtt{q}) \\ \mathtt{p} = \mathtt{q} &: PointsTo(\mathtt{p}, r) \leftarrow PointsTo(\mathtt{q}, r) \\ \mathtt{p} = {}^{\star}\mathtt{q} &: PointsTo(\mathtt{p}, r) \leftarrow PointsTo(\mathtt{q}, s), PointsTo(s, r) \\ {}^{\star}\mathtt{p} = \mathtt{q} &: PointsTo(r, s) \leftarrow PointsTo(\mathtt{p}, r), PointsTo(\mathtt{q}, s) \end{aligned}$$

It is well known that this algorithm can be implemented in time cubic in the number of variables. However, the rules do *not* directly suggest a cubic-time implementation. This issue was explored in Section 7.4.

Some authors [31], [7], [29], [16] have proposed algorithms to compute an approximation of Andersen's algorithm by storing *PointsTo* facts approximately but compactly. We discuss the representation used in [31] in Section 8.3 below.

1. As is customary in pointer analysis literature, the term "flow insensitive" is used here to also imply absence of "kill" in the data transfer functions. By contrast, we have defined FI as transformation that alters only the control, leaving the data transfer functions unchanged.

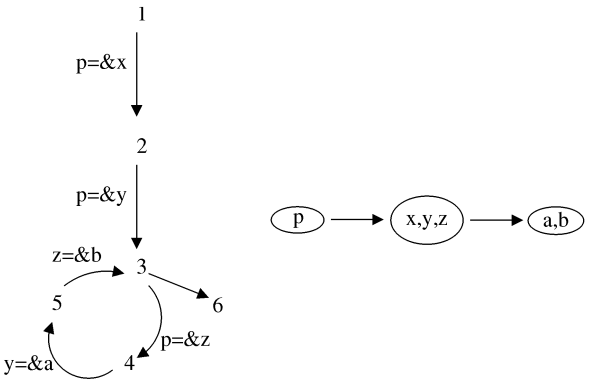


Fig. 12. A CFG and the storage shape graph computed from it using Steensgaard's algorithm.

8.2 Reachability Analysis

We now define each of the three algorithms just sketched as the reachability analysis of a transformed program. We assume below that program P is a pointer program over a points-to relation, obtained from a CFG by the translation of Section 4, and then transformed by the $\mathrm{Rel}_{Ptr}(\_)$ and $\mathrm{CSet}(\_)$ transformations of Section 5.

1. Given a program P, the HBCC flow-sensitive algorithm computes reachability analysis on $FI(LA(P))$ as it is based on data-flow analysis. Its complexity is $O(n^6)$, where n is the number of pointer variables. Note that data update functions in the points-to program model are not necessarily distributive. Consider $dnext$ for the statement ${}^{\star}\mathtt{s} = \mathtt{x}$ on two $Data$ elements $A = \{\mathtt{x} \rightarrow u,\ \mathtt{s} \rightarrow p\}$ and $B = \{\mathtt{x} \rightarrow v, \mathtt{s} \rightarrow q\}$. It can be seen that $dnext(A \cup B) > dnext(A) \cup\ dnext(B)$. Therefore, local accumulation (and data-flow analysis) are inexact, which is to be expected in a polynomial-time "solution" to this NP-hard problem.
2. Given a program P, the HBCC flow-insensitive algorithm computes reachability analysis on $FI(AC(P))$. Its complexity is $O(n^4)$; it runs faster but computes a less precise result than the previous case.
3. Given a program P, Andersen's algorithm computes reachability analysis on $MG(FI(AC(P)))$. It also uses the program transformation of Section 7.4 to compute the fixed point of the merged program more efficiently, in $O(n^3)$ time. Although it computes the same answer as the HBCC flow-insensitive algorithm, it runs faster by exploiting the structure of its statements.

8.3 Storage Shape Graphs

In Section 5, we discussed how pointer relationships can be represented as variable to value mappings and as points-to relations. Other representations are also used. Steensgaard's algorithm [31] takes a program as input and computes a *storage shape graph*, which is a conservative representation of the pointer relationships that can occur during program execution. Fig. 12 shows a CFG and the storage shape graph that Steensgaard's algorithm computes from it. One can

conclude from this graph that x cannot point to y during any program execution. A storage shape graph has the property that 1) each program variable is associated with at most one node of the graph, and 2) each node has at most one outgoing edge. (Note that [4] uses the term "storage shape graph" for representing the shape of heap-based data structures. In this paper, we use the term as Steensgaard does in [31], where the graph represents points-to relationship between scalar variables.)

Steensgaard's analysis can be defined as reachability analysis on a transformed version of the input program. To do this, first abstract the program so that its statements operate on a storage shape graph rather than a points-to relation. Next, apply the state accumulation and flow insensitivity transformations to the abstract program. To make the reachability analysis work, it is also necessary to interpret the state predicate $x \rightarrow y$ appropriately for storage shape graphs.

This scheme uses the state accumulation transformation, so we must show that programs that operate on storage shape graphs are monotonic.

First, we define an abstraction mapping from points-to relations to storage shape graphs. A shape graph can be represented as a pair $(\sim, \mapsto)$. The nodes of the graph are the equivalence classes of the equivalence relation $\sim$ on variables, and $\mapsto$ is the points-to relation on nodes. Our abstraction function α maps a points-to relation R to storage shape graph $(\sim, \mapsto)$ as follows: We define $\sim$ as the least relation satisfying

$$x \sim y = (x = y) \text{ or } (\exists u, w : u \sim w \text{ and } u \rightarrow x \text{ and } w \rightarrow y)$$

and, letting $[x]$ be the equivalence class of $\sim$ to which x belongs, we define that $[x] \mapsto [y]$ if there exists a u in $[x]$ and a w in $[y]$ such that $u \rightarrow w$. A semantics for pointer statements over storage shape graphs can be defined analogously to the semantics for pointer statements over points-to relations in Section 5.

Next, we show that the states of the program on storage shape graphs form a lattice. Let γ be a concretization function that maps a storage shape graph $(\sim, \mapsto)$ to the points-to relation R in which $x \rightarrow y \in R$ if $[x] \mapsto [y]$. Then, we can define an order relation $\leq$ on storage shape graphs by $g_1 \leq g_2 = \gamma(g_1) \subseteq \gamma(g_2)$ and a join operation $\vee$ on storage shape graphs by $g_1 \vee g_2 = \alpha(\gamma(g_1) \cup \gamma(g_2))$. (An implementation of $\vee$ would work directly on storage shape graphs for efficiency reasons.) The pointer statements on storage shape graphs are monotonic with respect to this lattice. Using these definitions, Steensgaard's algorithm computes $FI(AC(P))$, where P is defined on storage shape graphs.

Other program transformations can be applied to pointer programs on storage shape graphs. For example, one can derive a new algorithm that computes $FI(LA(P)))$. This algorithm will yield less precise results than Hind et al.'s algorithm, but has the possible advantage of a smaller state space because storage shape graphs are more compact than points-to relations. It is more precise than Steensgaard's, but incomparable to Andersen's analysis. We do not examine this algorithm in detail here; the purpose of mentioning it is to show how algorithms can be created easily by using our program transformations as basic building blocks.

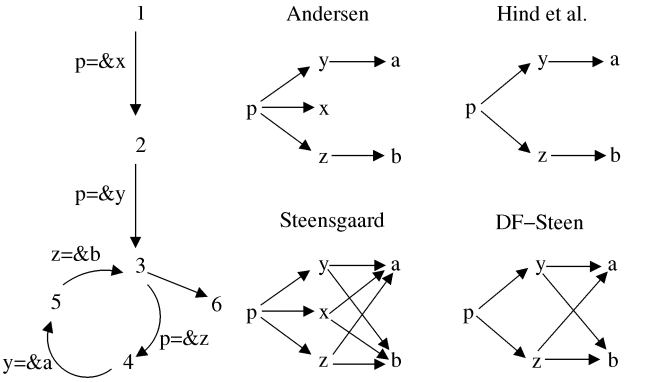


Fig. 13. Left: An example CFG. Right: Diagrams showing points-to relations at fixed point for control point 3 and various algorithms.

Fig. 13 shows a program for which, at control point 3, Andersen's, Hind's, Steensgaard's, and the algorithm just described (called DF-Steen in the figure) all give different points-to relations. In this figure, the storage shape graphs (for the Steensgaard and DF-Steen cases) are represented as points-to relations obtained by applying the concretization function γ above to the storage shape graphs.

9 Discussion

We defined a number of program transformations on a general program model and showed how efficient points-to analysis can be defined as the reachability analysis of a transformed pointer program. The program transformations we defined are general and can be used not only to recreate some of the existing points-to algorithms but also to devise new points-to algorithms. Section 8.3 contains an example of such an algorithm. Our program transformations can also be used outside of points-to analysis. For example, in Section 7.6, we showed how data flow analysis can be defined as the reachability analysis of a transformed program.

Since our transformations map a program to an abstract version of the program, it makes sense to ask whether they could be defined naturally through the framework of abstract interpretation [6]. To do so, we would capture the effect of our program transformations as abstract semantic interpretations and then, for correctness, show that the concrete and abstract meanings of a program are related via Galois connections. Some of the transformations we have defined fit well within the general framework of abstract interpretation. In deriving a program over relations from a program over environments (Section 5), we explicitly defined abstraction function α and concretization function γ and used the framework of abstract interpretation to prove the relationship between the original and derived programs. The role of abstract interpretation in deriving a program over storage shape graphs from a program over relations (Section 8.3) is similar.

The local accumulation transformation (Section 7.5) may be an interesting candidate for the application of abstract interpretation, but we did not use the framework of abstract interpretation for the results of Section 7.5. The transformations of control flow insensitivity, state accumulation, and

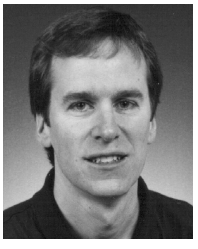
state merging can be defined using abstract interpretation, but doing so is uninteresting as the abstraction and concretization functions are just the identity function.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their detailed comments, which helped improve the presentation. They also thank the reviewers of a previous version that appeared in the 2002 Symposium on Foundations of Software Engineering. Dave Schmidt also provided several insightful comments on this work.

REFERENCES

[1] L. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD thesis, DIKU, Univ. of Copenhagen, May 1994.
[2] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis, "Property Preserving Simulations," *Proc. Fourth Int'l Workshop Computer Aided Verification,* pp. 260-273, 1992.
[3] V. Chakaravarthy, "New Results on the Computability and Complexity of Points-to Analysis," *Proc. 30th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* Jan. 2003.
[4] D.R. Chase, M. Wegman, and F.K. Zadeck, "Analysis of Pointers and Structures," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 296-310, 1990.
[5] E.M. Clarke, O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *Proc. 19th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 343-354, 1992.
[6] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. Fourth Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 238-252, 1977.
[7] M. Das, "Unification-Based Pointer Analysis with Directional Assignments," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 35-46, 2000.
[8] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond *k*-Limiting," *ACM SIGPLAN Notices,* vol. 29, no. 6, pp. 230-241, 1994.
[9] M. Emami, R. Ghiya, and L.J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 242-257, 1994.
[10] M. Fahndrich, J.S. Foster, Z. Su, and A. Aiken, "Partial Online Cycle Elimination in Inclusion Constraint Graphs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 85-96, 1998.
[11] R. Ghiya, D.M. Lavery, and D.C. Sehr, "On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 47-58, 2001.
[12] M.S. Hecht, *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.
[13] N. Heintze and O. Tardieu, "Ultra-Fast Alias Analysis Using CLA," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2001.
[14] N. Heintze and O. Tardieu, "Demand-Driven Pointer Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 24-34, 2001.
[15] M. Hind, M. Burke, P. Carini, and J. Choi, "Interprocedural Pointer Alias Analysis," *ACM Trans. Programming Languages and Systems,* vol. 21, no. 4, pp. 848-894, 1999.
[16] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" *2001 ACM SIGPLAN–SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 54-61, June 2001.
[17] S. Horwitz, "Precise Flow-Insensitive May-Alias Analysis is NP-Hard," *ACM Trans. Programming Languages and Systems,* vol. 19, no. 1, pp. 1-6, Jan. 1997.
[18] N.D. Jones and F. Nielson, "Abstract Interpretation: A Semantics-Based Tool for Program Analysis," *Handbook of Logic in Computer Science,* pp. 527-629, 1994.
[19] W. Landi, "Interprocedural Aliasing in the Presence of Pointers," PhD thesis, Rutgers Univ., Jan. 1992.
[20] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems,* vol. 1, no. 4, pp. 323-337, Dec. 1992.
[21] W. Landi and B.G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 235-248, 1992.
[22] W. Landi and B.G. Ryder, "Pointer-Induced Aliasing: A Problem Classification," *Proc. 18th Ann. ACM Symp. Principles of Programming Languages,* pp. 93-103, 1991.
[23] T.J. Marlowe and B.G. Ryder, "Properties of Data Flow Frameworks: A Unified Model," *Acta Informatica,* vol. 28, pp. 121-163, 1990.
[24] D. McAllester, "On the Complexity Analysis of Static Analyses," *Proc. Static Analysis Symp.,* 2001.
[25] K.S. Namjoshi and R.P. Kurshan, "Syntactic Program Transformations for Automatic Abstraction," *Proc. 12th Conf. Computer Aided Verification,* E.A. Emerson and A.P. Sistla, eds., 2000.
[26] F. Nielson, H. Nielson, and C. Hank, *Principles of Program Analysis: Flows and Effects.* Springer, 1999.
[27] T. Reps, "Program Analysis via Graph Reachability. Information and Software Technology," *Information and Software Technology,* vol. 40, nos. 11-12, pp. 701-726, 1998.
[28] D. Schmidt, "Data-Flow Analysis is Model Checking of Abstract Interpretations," *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 1998.
[29] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-to Analysis," *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 1997.
[30] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Dataflow Analysis," *Program Flow Analysis: Theory and Applications,* S.S. Muchnick and N.D. Jones, eds., 1980.
[31] B. Steensgaard, "Points-to Analysis in Almost Linear Time," *Proc. 23th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 1996.
[32] B. Steffen, "Generating Data-Flow Analysis Algorithms for Modal Specification," *Science of Computer Programming,* vol. 21, pp. 115-139, 1993.



Glenn Bruns received the PhD degree in computer science from the University of Edinburgh, Scotland, in 1996. He is a member of the technical staff in the Computing Sciences Research Center at Bell Labs, Lucent Technologies. His main research interests are in model checking, temporal logic, concurrency theory, and automated software development. From 1985 to 1989, he was a member of the technical staff in the Software Technology Program at MCC in Austin, Texas. From 1990 to 1996, he was a senior research fellow in the Laboratory for Foundations of Computer Science at the University of Edinburgh. He is the author of *Distributed Systems Analysis with CCS* (Prentice Hall, 1997).

Satish Chandra received the PhD degree from the University of Wisconsin-Madison in 1997 and the BTech degree from the Indian Institute of Technology-Kanpur in 1991, both in computer science. From 1997 to 2002, he was a member of the technical staff in the Computing Sciences Research Center at Bell Laboratories, where his research focused on program analysis, domain-specific languages, and data-communication protocols. In September 2002, he joined IBM's India Research Laboratory, where he works on program analysis and distributed computation. He is a member of the ACM, the IEEE, and the IEEE Computer Society.