Searching for Points-To Analysis

Glenn Bruns and Satish Chandra
Bell Laboratories, Lucent Technologies
2701 Lucent Lane, Lisle, IL 60532
{grb,chandra}@research.bell-labs.com

ABSTRACT

The complexity of points-to analysis is well understood, but the approximations used to carry out points-to analysis efficiently are less well understood. In this paper we characterize points-to analysis as a reachability problem on a program's state space. Reachability analysis can be performed approximately but more efficiently for a program to which certain basic program transformations have been applied. We show the source of approximation and efficiency in several existing points-to analysis algorithms in terms of these generic program transformations.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; F.3.1 [**Theory of Computation**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*

General Terms

Languages,Theory,Verification

1. INTRODUCTION

Points-to analysis is an established topic in program analysis. Algorithms that carry out points-to analysis efficiently on large programs have been developed [8, 5], and some commercial compilers incorporate optimizations that rely on points-to analysis [7]. Also, points-to analyses have been formalized in the frameworks of iterative data-flow analysis [12], set constraints [6], graph reachability [17], and logic programming [19].

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 labelled 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 [10]. 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 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.

The paper is organized as follows. In the next section we define our program model and how programs can be interpreted as LTSs. In Section 3, we define pointer programs in this framework. In the core of the paper, Section 4, we define five program transformations: flow-insensitivity, state accumulation, program merging, local accumulation, and data-flow approximation. In Section 5, we briefly describe three existing algorithms for points-to analysis and show how these can be seen as reachability analyses of transformed programs. Finally, in Section 6 we relate our approach to abstract interpretation and mention other approximations not dealt with directly in the paper.

2. PROGRAM MODEL

We assume a set *Guard* of boolean expressions and a set *Cmd* of commands. A *statement* is then an element of

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

$Guard \times Cmd$ and a *program* is a set of statements. For example, a statment might have the form $(gtzero(i), decr(j))$, informally meaning that that j should be decremented if $i > 0$.

We define the meaning of programs in two steps. First, we interpret statements locally by interpreting guards as predicates and commands as state update functions. We assume a set *State* of states, and let $Pred = State \rightarrow Bool$ be the set of state predicates and $Update = State \rightarrow State$ be the set of state update functions. We then assume an interpretation function $G : Guard \rightarrow Pred$ for guards and an interpretation function $C : Cmd \rightarrow Update$ for commands. Given a program $P = \{(g_1, c_1), \ldots, (g_n, c_n)\}$, we define its interpreted form $[P]$ as $\{(en_1, f_1), \ldots, (en_n, f_n)\}$, where $en_i = G(g_i)$ and $f_i = C(c_i)$.

We will define our program transformations directly on objects of the form $\{(en_1, f_1), \ldots, (en_n, f_n)\}$. We ambiguously call these objects *programs* and use variable P to stand for them.

Having interpreted statements, we then define the global meaning of a program as a labelled transition system (LTS). An LTS is a directed graph in which a node represents a program state and an edge represents the effect of a program statement. Nodes are labelled with atomic propositions that capture basic properties of program state. The particular atomic propositions used to label states depends on the application.

DEFINITION 1. *Let $P = \{(en_1, f_1), \ldots, (en_n, f_n)\}$ be a program. The LTS for P over State, written $(P, State)$, has State as its node set and a transition relation defined as follows:*

$$s \xrightarrow{(en,f)} s' \triangleq \exists (en, f) \in P.en(s) \text{ and } s' = f(s)$$

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

2.1 Modelling Control-Flow Graphs

A *control-flow graph* (CFG) is a directed graph in which nodes represent program statements and edges represent control flow between statements. Every CFG contains distinguished *entry* and *exit* nodes. As is common in flow analysis, we assume that if a program has just executed the statement at a certain node then the statement at any node along an outgoing edge can be executed next; i.e., branch conditions are replaced by a non-deterministic choice.

We capture CFGs in a specialized form of our program model, in which each state has a control part and a data part. We assume a set CP of control points and a set *Data* of data values, and define $Control = 2^{CP}$ and $State = Control \times Data$. (We define the control component of a state as a set of control points because we will later "join" states.) For syntax, we assume a set *DExpr* of data update expressions and define the set *Guard* to be all expressions of the form $at(m)$, where $m \in Control$, and set *Cmd* to be all expressions of the form $(to(m), expr)$, where $m \in Control$ and $expr \in DExpr$. To interpret data expressions, we assume a function $D : DExpr \rightarrow (Data \rightarrow Data)$. Then we define interpretation functions G and C by

$$\begin{aligned} G(at(m)) &= \lambda(c, x).(m \in c) \\ C((to(m), expr)) &= \lambda(c, x).(\{m\}, D(expr)(x)) \end{aligned}$$

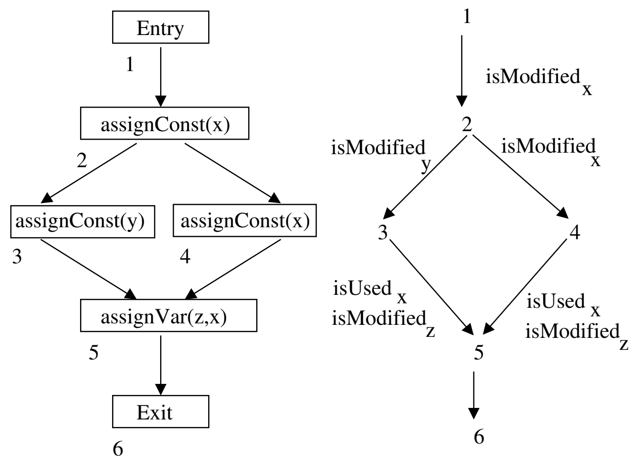

Figure 1: A CFG and its representation as an uninterpreted LTS.

In writing CFG programs in their semantic form, we will often write a state update function as a pair (f_c, f_d), where $f_c \in Control \rightarrow Control$ and $f_d \in Data \rightarrow Data$. The state update function denoted by (f_c, f_d) maps (c, x) to $(f_c(c), f_d(x))$.

To map a CFG into this program model, suppose a CFG has an edge from node n to node m, and that statement s is at node m. Then the program for the CFG has a corresponding statement $(at(n), (to(m), s))$. In this translation, a control point value of m means that the statement at CFG node m just been executed. We refer to a CFG captured in our program model as a *CFG program*.

For example, consider the CFG on the left of Figure 1. The program corresponding to this CFG would contain a statement $(at(2), (to(3), assignConst(y)))$ corresponding to the transition from node 2 to 3 in the CFG.

2.2 Program Analysis

We consider only the reachability analysis of programs. 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 2. *Let $(P, State)$ be an LTS with s in State. Then $s, (P, State) \models EF\phi$ holds iff there exists an s' in State such that $s \rightsquigarrow s'$ and $\phi(s')$.*

Reachability can also be expressed as a fixed-point property. The function $f(A) \triangleq \{s\} \cup \{s'' \mid \exists s' \in A.s' \rightarrow s''\}$ is a monotonic function on sets of states. Its least fixed-point μf is the set of states reachable from s. So $s, (P, State) \models \mathrm{EF}\phi$ just if there exists a state s' in μf such that $\phi(s')$. By Knaster-Tarski, a least fixed-point for f must exist because the subsets of *State*, ordered by $\subseteq$, form a complete lattice.

We emphasize that the role of the LTS here 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 the only storage required is that 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.

An LTS $(P, State)$ could be called an "interpreted" LTS, as it represents all possible program executions. An alternative for program analysis is to use an "uninterpreted" LTS in which the nodes represent program control points (e.g., see [21] and [18]), and edges are labelled with program actions (see the LTS on the right of Figure 1). For example, Schmidt shows [18] how the property of variable liveness can be expressed as the following formula of the modal mu-calculus:

$$\text{isLive}(x) = \mu Z.\langle \text{isUsed}_x \rangle \text{true} \vee \langle -\text{isModified}_x \rangle Z$$

The formula states that variable x is live at a node if there exists an outgoing transition labeled with action isUsed_x, or, if there exists an outgoing transition, not labelled with action isModified_x, leading to a node in which $\text{isLive}(x)$ holds. The operator μ is a least fixpoint operator; here it is used to force $\text{isLive}(x)$ to be false when no isUsed_x transition can ever be reached.

A points-to analysis problem can be captured as a modal mu-calculus formula on an uninterpreted LTS, but for exact points-to analysis the formula is complicated, and the size of the formula grows exponentially in the number of program variables. In what follows we use only interpreted LTSs, and refer to them as just LTSs.

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 [13].

DEFINITION 3. *Given a CFG, the* precise points-to problem *is to determine whether after a given node n, a given variable p points to another variable q along some path from* `Entry` *to n.*

We make the following assumptions about pointer programs: 1) The program consists of a single procedure – if there are multiple procedures, we treat calls and returns essentially as goto's within one large procedure; 2) The program does not allocate memory dynamically, as in the presence of dynamic memory allocation, points-to analysis is known to be undecidable; 3) All program variables are scalar variables and not records; 4) Pointer statements come in only four syntactic forms: `p = &q`, `p = q`, `p = *q`, and `*p = q`. We assume that statements with multiple `*`'s in front of variable names have been rewritten to these four cases by introducing temporary variables. This reduction is not precise for the flow-insensitive case [11].

Notwithstanding these assumptions, the precise points-to analysis problem is NP-hard for programs with multi-level pointers [13]. By multi-level pointers, we mean that pointers can contain addresses of pointer variables, which is the case in general. Practical polynomial-time pointer analysis algorithms for C-like languages compute approximate answers to the points-to problem.

3.1 Program Representation

The points-to analysis problem is defined on control-flow graphs, so we capture these graphs as CFG programs. Syntactically, we assume that the set *DExpr* of expressions consists of *assignAddr(x,y)*, *assign(x,y)*, *assignStar(x,y)* and *starAssign(x,y)*, where x and y range over program variables. For brevity, we write `p = *x` for *assignStar(*`p`,`x`*)*, and similarly for the other pointer statements.

Semantically, we let the elements of *Data* be points-to relations R on program variables. We write $p \rightarrow q$ rather than (p, q) when working with points-to relations. The interpretation of each pointer statement is given below, where $R \setminus p$ stands for the set $R - \{p \rightarrow r \mid p \rightarrow r \in R\}$.

$$\begin{aligned}
(\texttt{p = \&q})(R) &= R \setminus \texttt{p} \cup \{\texttt{p} \rightarrow \texttt{q}\} \\
(\texttt{p = q})(R) &= R \setminus \texttt{p} \cup \{\texttt{p} \rightarrow r \mid \texttt{q} \rightarrow r \in R\} \\
(\texttt{p = *q})(R) &= R \setminus \texttt{p} \cup \{\texttt{p} \rightarrow r \mid \exists s.\{\texttt{q} \rightarrow s, s \rightarrow r\} \subseteq R\} \\
(\texttt{*p = q})(R) &= \text{let } L = \{r \mid \texttt{p} \rightarrow r \in R\} \text{ in} \\
&\quad\; (R - \mathit{kill}(L, R)) \cup \\
&\quad\; \{r \rightarrow s \mid r \in L \text{ and } \texttt{q} \rightarrow s \in R\})
\end{aligned}$$

Note that for `p = *q`, with `q` a null pointer, the statement has no effect on *Data*, as opposed to an abort in a C program execution. The function for `*p = q` uses an auxiliary function *kill* that removes the pairs from R that the statement will certainly override. If $|L| = 1$, then p points to exactly one other variable r, so *kill* removes all pairs of the form $r \rightarrow \_$ (we use $\_$ as wildcard) from R: $\mathit{kill}(L, R) = R \setminus r$. If $|L| > 1$, we cannot remove all pairs of the form $r \rightarrow \_$ because we do not know whether p *must* point to r in that execution: $\mathit{kill}(L, R) = \{\}$. If $|L| = 0$, p is a null pointer, and the statement would abort in a C program execution, but by convention [10] we remove all alias pairs from R: $\mathit{kill}(L, R) = R$. With this definition for *kill* function f_d is monotonic.

Example: Consider the following simple CFG:

`Entry`	
`1: x = &y;`	$\{\texttt{x} \rightarrow \texttt{y}\}$
`2: z =  x;`	$\{\texttt{x} \rightarrow \texttt{y}, \texttt{z} \rightarrow \texttt{y}\}$
`3: x = &w;`	$\{\texttt{x} \rightarrow \texttt{w}, \texttt{z} \rightarrow \texttt{y}\}$
`Exit`	

Pointer relationships that hold after execution of each statement are shown to the right. P and $[P]$ for this CFG are shown in Table 1.

3.2 Analysis using LTSs

The nodes of an LTS derived from a pointer program are states of the form (cp, R), where cp is a control point set and R is a points-to relation. For example, the LTS for our example program can be written as:

$$(\{\texttt{Entry}\}, \emptyset) \xrightarrow{stmt_1} (\{1\}, \{\texttt{x} \rightarrow \texttt{y}\}) \xrightarrow{stmt_2}$$
$$(\{2\}, \{\texttt{x} \rightarrow \texttt{y}, \texttt{z} \rightarrow \texttt{y}\}) \xrightarrow{stmt_3} (\{3\}, \{\texttt{x} \rightarrow \texttt{w}, \texttt{z} \rightarrow \texttt{y}\})$$

The LTS nodes are labelled with sets of atomic propositions of the form $x \rightarrow y$ and $cp = i$; the set of atomic propositions at node (cp, R) contains $x \rightarrow y$ if $(x, y) \in R$ and contains $cp = i$ if the value of $i \in cp$. The precise points-to problem of Section 3 can then be expressed as a reachability problem.

DEFINITION 4. *Let P be a CFG program. The precise points-to problem of whether $p \rightarrow q \in R$ holds after node n can be written*

$$(\{\mathtt{Entry}\}, \emptyset), (P, \mathit{State}) \models EF(cp = n \wedge p \rightarrow q)$$

For our example program, the answer to the precise points-to problem of whether `z` points to `y` after statement 2 is yes, because of the path $(\{\texttt{Entry}\}, \emptyset) \rightsquigarrow (\{2\}, \{\texttt{x} \rightarrow \texttt{y}, \texttt{z} \rightarrow \texttt{y}\})$.

guard	command	$en(cp)$	$f_c(cp)$	$f_d(R)$
$at(\texttt{Entry})$	$to(1)$, `x = &y`	$\texttt{Entry} \in cp$	$\{1\}$	$R \setminus \texttt{x} \cup \{\texttt{x} \to \texttt{y}\}$
$at(1)$	$to(2)$, `z = x`	$1 \in cp$	$\{2\}$	$R \setminus \texttt{z} \cup \{\texttt{z} \to p \mid \texttt{x} \to p \in R\}$
$at(2)$	$to(3)$, `x = &w`	$2 \in cp$	$\{3\}$	$R \setminus \texttt{x} \cup \{\texttt{x} \to \texttt{w}\}$

Table 1: P and $[P]$ for our example CFG.

4. PROGRAM TRANSFORMATIONS

In this section we define some basic program transformations that can be used to derive existing points-to analysis algorithms. We first present control-flow insensitivity (Section 4.1), state accumulation, (Section 4.2), program merging, (Section 4.3), and local accumulation (Section 4.5) as independent transformations. We use the examples of Figure 2 to illustrate the transformations. We also show how a transformation used in logic programming can be used to increase efficiency (Section 4.4), and that data-flow analysis (Section 4.6) can be defined using local accumulation plus control-flow insensitivity.

4.1 Control ¤ow insensitivity

A program analysis is said to be *flow-insensitive* if the order of program statements is ignored. In terms of our program model, flow insensitivity means all statements are always enabled.

Definition 5. *Let $P = \{(en_1, f_1), \ldots, (en_n, f_n)\}$ be a program. Then the* flow-insensitive *version $FI(P)$ of P is $\{(en, f_1), \ldots, (en, f_n)\}$, where $en(s) \triangleq true$.*

The flow-insensitive version of the example program of Section 3.1 is as follows:

$en(cp)$	$f_c(cp)$	$f_d(R)$
$true$	$\{1\}$	$R \setminus \texttt{x} \cup \{\texttt{x} \to \texttt{y}\}$
$true$	$\{2\}$	$R \setminus \texttt{z} \cup \{\texttt{z} \to p \mid \texttt{x} \to p \in R\}$
$true$	$\{3\}$	$R \setminus \texttt{x} \cup \{\texttt{x} \to \texttt{w}\}$

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

Proposition 1. *Let P be a program over State, with $s \in$ State, and ϕ be a predicate on State. Then*

$$s, (P, State) \models EF\phi \;\Rightarrow\; s, (FI(P), State) \models EF\phi$$

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, 3, 15]. 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 [3]. Furthermore, if the abstracted LTS satisfies a temporal logic formula of a particular class, then so does the original LTS [2].

4.2 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 source state. We might hope to find a particular pointer relationship more quickly by redefining statements so that the pointer relation always increases when travelling from a state to a successor state.

To formalize this idea, we assume the elements of state form a complete lattice $(State, \leq)$. We require only that $(State, \leq)$ be a partial order with joins and a bottom element $\perp$, but for simplicity we specify a lattice. In all that follows we assume that $(State, \leq)$ is a complete lattice.

Definition 6. *Let $(State, \leq)$ be a 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 $f : State \to State$ is* monotonic *if $s \leq s'$ implies $f(s) \leq f(s')$.*
3. *A statement (en, f) is monotonic if en is up-closed and f is monotonic.*
4. *A program is monotonic if all its statements are.*
5. *A statement (en, f) is* accumulative *if $s \leq f(s)$ for all s in State.*
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 $f(x)$ be: if $x = 0$ then 2 else x. Then f is accumulative but not monotonic.

Definition 7. *Let $P = \{(en_1, f_1), \ldots, (en_n, f_n)\}$ be a program. Then the* accumulated *version $AC(P)$ of P is $\{(en_1, f'_1), \ldots, (en_n, f'_n)\}$, where $f'_i(s) \triangleq s \vee f_i(s)$.*

For CFG programs, the set $State = Control \times Data$ of program states is a complete lattice as long as $Data$ is a complete lattice. This is so because $Control$ naturally forms a complete lattice under set inclusion, and the product of two complete lattices is a complete lattice. The join operation for $State$ is then $(c_1, x_1) \vee (c_2, x_2) = (c_1 \cup c_2, x_1 \vee x_2)$. We will assume in all that follows that CFG programs operate on data in a complete lattice.

For points-to analysis, $Data$ is a points-to relation, join is set union, and $Data$ is ordered by set inclusion. The four pointer statements of our example are all monotonic with respect to this order.

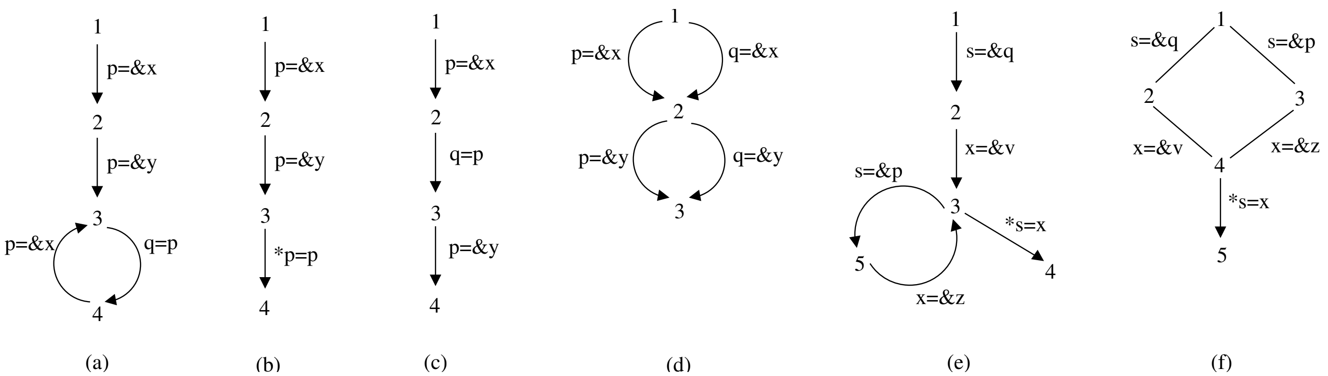

Figure 2: Example CFGs. CFGs (e) and (f) are adapted from Landi's thesis.

The accumulated, flow-insensitive version of our example program is as follows:

$en(cp)$	$f_c(cp)$	$f_d(R)$
true	$cp \cup \{1\}$	$R \cup \{\mathtt{x} \to \mathtt{y}\}$
true	$cp \cup \{2\}$	$R \cup \{\mathtt{z} \to p \mid \mathtt{x} \to p \in R\}$
true	$cp \cup \{3\}$	$R \cup \{\mathtt{x} \to \mathtt{w}\}$

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

$$s, (P, \mathit{State}) \models \mathit{EF}\phi \;\Rightarrow\; s, (\mathit{AC}(P), \mathit{State}) \models \mathit{EF}\phi$$

In making a program accumulative one loses precision. Figure 2(a) shows a CFG program for which the LTS of the accumulated version (Figure 4(ii)) has fewer states than the LTS of the original program (Figure 4(i)).

Flow insensitivity and accumulation are independent transformations. Figure 2(b) shows a CFG program for which accumulating the flow-insensitive version loses precision compared to just the flow-insensitive version. The former produces $x \to y$ (see Figure 4(iii)) while the latter does not. By contrast, Figure 2(c) 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.

4.3 Program 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 8. *Let $P = \{(en_1, f_1), ..., (en_n, f_n)\}$ be a program. Then the* merged *version $MG(P)$ of P is $\{(en, f)\}$, where*

$$en(s) \;\triangleq\; \bigvee\{en_i(s) \mid i \in \{1, \dots, n\}\}$$

$$f(s) \;\triangleq\; \bigvee\{f_i(s) \mid en_i(s) \text{ and } i \in \{1, \dots, n\}\}$$

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*.

The merged, accumulated, flow-insensitive version of our example program is as follows:

$en(cp)$	$f_c(cp)$	$f_d(R)$
true	$cp \cup \{1, 2, 3\}$	$R \cup \{\mathtt{x} \to \mathtt{y}\} \cup$ $\{\mathtt{z} \to p \mid \mathtt{x} \to p \in R\} \cup \{\mathtt{x} \to \mathtt{w}\}$

Figure 2(d) shows a CFG program for which the LTS of the merged version (Figure 4(v)), the accumulative version (Figure 4(vi)), and the merged, accumulative version (Figure 4(vii)) are all different; merging reduces the number of states in the LTS. Figure 4(iv) shows the LTS of the original program.

We would like to say that program errors can be discovered by considering the merged form of a program. However, some errors can be found in normal program execution but not merged execution. Suppose our states are $(\{0, 1\} \times \{0, 1\}, \leq)$, where $(x, y) \leq (x', y')$ if $x' \geq x$ and $y' \geq y$, and that $f(x, y) = (0, 1)$ and $g(x, y) = (1, 0)$. Through merged execution from $(0, 0)$ we reach $(f \vee g)(0, 0) = (1, 1)$, which is unreachable through normal execution. However, if we require that statements be accumulative and that if a state satisfies an interesting property then so do all greater states, then we can show that a merged execution can find interesting properties if and only if some normal execution can.

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

$$s, (P, \mathit{State}) \models \mathit{EF}\phi \;\Rightarrow\; s, (\mathit{MG}(P), \mathit{State}) \models \mathit{EF}\phi$$

If P is accumulative and ϕ is up-closed, then the converse also holds. 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 $(\mathit{State}, \leq)$, $s \in \mathit{State}$, and ϕ is up-closed. Then*

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

4.4 Computing State Ef£ciently

The merged, accumulated, flow-insensitive version of our example program has the form $\{(true, f)\}$, where f 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 f repeatedly until a fixed point is reached (a fixed point is guaranteed if *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 `x = *y`. Intuitively, applying the statement involves adding $\mathtt{x} \to w$ to points-to relation R if there exist variables z and w such that $\mathtt{y} \to z$ and $z \to w$ belong to R. The merged, accumulative, flow-insensitive version of the program will have f_d of the form:

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

A straightforward implementation of this function would, for the clause of f_d representing `x = *y`, iterate over every element $\mathtt{y} \to z$ of R, then search for an element $z \to w$ 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 $u = *w$ in the program, giving an overall running time for f_d of $O(n^4)$.

We now transform f_d to f'_d, 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.

$$\begin{aligned} f'_d((R, S)) = \\ &(R \cup \{\mathtt{x} \to w \mid \exists z. \mathtt{x} \hookrightarrow z \in S \text{ and } z \to w \in R\} \cup \ldots, \\ &\ S \cup \{\mathtt{x} \hookrightarrow z \mid \mathtt{y} \to z \in R\} \cup \ldots) \end{aligned}$$

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 f'_d 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 f_d and f'_d 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 f_d and f'_d as Datalog programs and using Theorem 1 of [14]. The idea of expressing points-to analysis as a logic program is not new. In [19], Shapiro and Horwitz give a Datalog formulation of the points-to analysis problem. In [17] Reps shows how the problem can be formulated as a CFL-reachability problem via a chain-form Datalog program. In [9] Heintze gives a demand-driven 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.

4.5 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 data value 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.

To formalize local accumulation we define *State* to be $Control \times (Control \to Data)$ instead of $Control \times Data$. Given a mapping $M : Control \to Data$ that labels control points with data values, we write $M[c := x]$ for the mapping that is like M except that c is mapped to x. Also, we define the mapping $M_\perp$ by $M_\perp(c) = \perp$ for all c in $Control$. Recall from Section 4.2 that *Data* and thus *State* are complete lattices.

DEFINITION 9. *Let P be a CFG program over $Control \times Data$. The locally-accumulative version $LA(P)$ of P is a program P' over $Control \times (Control \to Data)$ such that for every statement $(at(c), (to(c'), f_d))$ in P there is a statement $(at(c), (to(c'), f'_d))$ in P', where*

$$f'_d(M) = M[c' := M(c') \vee f_d(M(c))]$$

M is a *memo table* in the terminology of Schmidt [18].

Table 2 shows the locally-accumulative version of our example program. No control point is repeated along the only path, so in this example the locally-accumulative version is the same as the original. Figure 2(e) 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 $p \to v$ 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 $\phi(c, M) = \phi(c, M(c))$.

$en(cp)$	$f_c(cp)$	$f'_d(M)$
$\texttt{Entry} \in cp$	$\{1\}$	$M[1 := M(1) \cup M(\texttt{Entry}) \setminus \texttt{x} \cup \{\texttt{x} \rightarrow \texttt{y}\}]$
$1 \in cp$	$\{2\}$	$M[2 := M(2) \cup M(1) \setminus \texttt{z} \cup \{\texttt{z} \rightarrow p \mid \texttt{x} \rightarrow p \in R\}]$
$2 \in cp$	$\{3\}$	$M[3 := M(3) \cup M(2) \setminus \texttt{x} \cup \{\texttt{x} \rightarrow \texttt{w}\}]$

Table 2: Locally-accumulative version of the example program.

PROPOSITION 2. *Suppose P is a CFG program over State = Control × Data such that f_d in every statement $(at(c), (to(c), f_d))$ of P is monotonic, $(c, x) \in State$, and ϕ is up-closed. Then*

$$(c, x), (P, State) \models EF\phi \;\Rightarrow\; (c, M_\perp[c := x]), (LA(P), State') \models EF\phi$$

A function f is *distributive* if $f(a \vee b) = f(a) \vee f(b)$. A predicate ϕ is *existential* if $\phi(x \vee y) \Leftrightarrow \phi(x)$ or $\phi(y)$.

THEOREM 4. *In addition to the conditions of Prop. 2, let each f_d of P be distributive and ϕ be existential. Then*

$$(c, x), (P, State) \models EF\phi \;\Leftrightarrow\; (c, M_\perp[c := x]), (LA(P), State') \models EF\phi$$

4.6 Data Flow Approximation

Given a CFG program P over $State = Control \times Data$, *data-flow analysis* [4, 16] of P computes the result of the following metaprogram. Its termination is ensured if $Data$ is finite and every f_d is monotonic.

> **global** $M : Control \rightarrow Data$
> initially, $\forall i, M(i) = \perp$
> **repeat** until no change to M
>   pick a statement $(at(c), (to(c'), f_d))$
> $M(c') = M(c') \vee f_d(M(c))$
> **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.

THEOREM 5. *Let P be a monotonic CFG program over State = Control × Data, M be the result of data-flow analysis on P, and ϕ be up-closed. Then*

$$(\{\textit{Entry}\}, \perp), (FI(LA(P)), State') \models EF\phi \;\Leftrightarrow\; \exists i : \phi(i, M(i))$$

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 3 shows the flow-insensitive, locally-accumulative version of our example program. Again, since no control point is repeated, the result is the same as the original. Figure 2(f) shows a CFG program for which data-flow analysis gives a less precise result than local accumulation: the former produces $q \rightarrow z$ and $p \rightarrow v$, while the latter does not.

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. *Let P be a monotonic CFG program over State = Control × Data in which each f_d of P is distributive, and ϕ be up-closed and existential. Then*

$$(\{\textit{Entry}\}, \perp), (P, State) \models EF\phi \;\Leftrightarrow\; (\{\textit{Entry}\}, \perp), (FI(LA(P)), State') \models EF\phi$$

Data-flow analysis is also related to uninterpreted LTSs. Schmidt [18] 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, but in general it may compute a safe approximation.

5. POINTS-TO ANALYSIS AS REACHABILITY ANALYSIS

5.1 Existing Algorithms

We describe briefly the intra-procedural versions of two algorithms for computing safe but approximate answers to the precise points-to problem.

HBCC algorithms

We sketch the algorithm given in Hind, Burke, Carini and Choi [10]. Our presentation uses points-to sets, 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 $f_d(In[p])$, using the definition of f_d (see Section 3.1) applicable to p. For each statement, *MustKill* and *Gen* can be gleaned from the structure of the corresponding f_d.

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

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

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

$en(cp)$	$f_c(cp)$	$f'_d(M)$
true	$cp \cup \{1\}$	$M[1 := M(1) \cup M(\texttt{Entry}) \setminus \texttt{x} \cup \{\texttt{x} \rightarrow \texttt{y}\}]$
true	$cp \cup \{2\}$	$M[2 := M(2) \cup M(1) \setminus \texttt{z} \cup \{\texttt{z} \rightarrow p \mid \texttt{x} \rightarrow p \in R\}]$
true	$cp \cup \{3\}$	$M[3 := M(3) \cup M(2) \setminus \texttt{x} \cup \{\texttt{x} \rightarrow \texttt{w}\}]$

Table 3: Flow-insensitive, Locally-accumulative version of the example program.

final set of facts is the answer for all program points.

$$\begin{array}{rcl} \texttt{p = \&q} & : & \mathit{PointsTo}(\texttt{p},\texttt{q}) \\ \texttt{p = q} & : & \mathit{PointsTo}(\texttt{p},r) \leftarrow \mathit{PointsTo}(\texttt{q},r) \\ \texttt{p = *q} & : & \mathit{PointsTo}(\texttt{p},r) \leftarrow \mathit{PointsTo}(\texttt{q},s),\ \mathit{PointsTo}(s,r) \\ \texttt{*p = q} & : & \mathit{PointsTo}(r,s) \leftarrow \mathit{PointsTo}(\texttt{p},r),\ \mathit{PointsTo}(\texttt{q},s) \end{array}$$

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 4.4.

Many authors [20, 5, 19] have proposed algorithms to compute an approximation of Andersen's algorithm by storing *PointsTo* facts approximately but compactly. Since this approximation is very specific to *PointsTo* facts, we do not consider it here.

5.2 Reachability Analysis

We now define each of the three algorithms just sketched as the reachability analysis of a transformed program.

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 f_d's in the points-to program model are not necessarily distributive. Consider f_d for the statement `*s = x` on two $Data$ elements $A = \{\texttt{x} \rightarrow u, \texttt{s} \rightarrow p\}$ and $B = \{\texttt{x} \rightarrow v, \texttt{s} \rightarrow q\}$. It can be seen that $f_d(A \cup B) > f_d(A) \cup f_d(B)$. 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 4.4 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.

6. DISCUSSION

6.1 Abstract Interpretation

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 [4]. To do so we 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.

The transformations of control flow insensitivity, state accumulation, and program merging fit poorly into the classical framework of abstract interpretation because this framework depends on the notion of control point, and these transformations are defined on our basic program model, which has no notion of control point. However, these transformations can be defined using an abstract interpretation-like approach. We now sketch the basic idea. Consider the transformation of control flow insensitivity. We begin by defining a static (i.e., "collecting") semantic function for our standard semantics:

$$Int_c(A) \triangleq \{f(s) \mid s \in A \text{ and } (en, f) \in P \text{ and } en(s)\}$$

Given a set of states, this function gives the successors of these states. Then, to model control flow insensitivity, we define an abstract semantic function:

$$Int_a(A) \triangleq \{f(s) \mid s \in A \text{ and } (en, f) \in P\}$$

Both the concrete semantics and abstract semantics operate on sets of states. Furthermore, the mapping between the concrete and abstract worlds is trivial; we used the identify function on sets of states for the abstraction and concretization functions. We need to show that if a state is reachable under the concrete semantics then it is reachable also in the abstract semantics. To do this we only need to show that $Int_c(A) \subseteq Int_a(A)$ for all sets A of states.

The same approach can be used for state accumulation and program merging. However, in all of these cases the identity function is used for both abstraction and concretization, so the use of abstract interpretation is uninteresting.

The local accumulation transformation is more interesting to consider because it is based on the CFG program model, which does include a notion of control point. Here we can define a static semantic function that is similar to that of Section 4 of [4]. We define an *environment* as an element of $Data$ rather than as a mapping from variables to values, as in our CFG model there is no notion of variable. In the case of pointer analysis, an environment is a set of pointer relations. The static semantics associates each control point with a *context*, which is a set of environments. The abstract semantics is obtained by modifying the static semantics slightly to capture local accumulation.

However, we seem to gain little from attempting to view this transformation as an abstract interpretation because again the abstraction and concretization functions are just the identify function. We have simply reframed our abstraction ideas as alternative program semantics rather than program transformations.

6.2 Data abstraction

For pointer programs we have defined $Data$ as the set of points-to relations, but there are alternatives. Consider Steensgaard's algorithm [20], which computes a *storage shape graph* in which nodes represent sets of program variables and

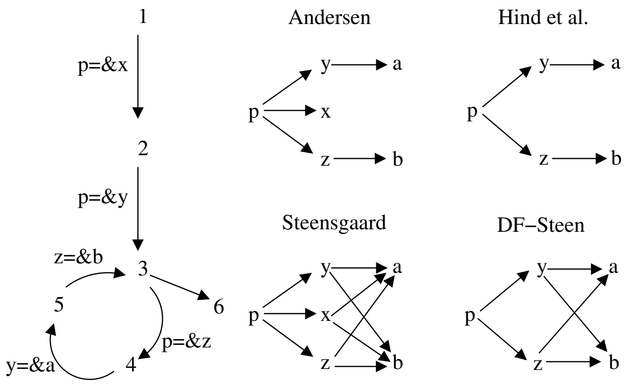

Figure 3: Left: an example CFG. Right: diagrams showing points-to relations at fixed point for control point 3 and various algorithms.

edges represent pointer relationships. The graph has the property that (a) each program variable is associated with at most one node of the graph, and (b) each node has at most one outgoing edge.

In the style of abstract interpretation, we can define abstraction and concretization functions between points-to relations and storage shape graphs. For example, the concretization function γ maps a storage shape graph g to a points-to relation R in which $x \rightarrow y \in R$ if there is an edge A_1 to A_2 in g such that $x \in A_1$ and $y \in A_2$. Then we can define an order relation $\leq'$ on $Data'$ by $g_1 \leq' g_2 \triangleq \gamma(g_1) \subseteq \gamma(g_2)$, and a join operation $\vee'$ on $Data'$ by $g_1 \vee' g_2 \triangleq \alpha(\gamma(g_1) \cup \gamma(g_2))$. (An implementation would compute join directly on $Data'$ for efficiency reasons.) With these definitions, Steensgaard's algorithm computes $AC(FI(P))$, where P is defined on $Data'$. The functions f_d can be defined analogously to the definitions in Section 3.1.

It is clear that other program transformations can be applied to P defined on $Data'$. For example, one can derive a new points-to algorithm that computes $FI(LA(P)))$. This algorithm will yield less precise results that Hind *et al*'s algorithm, but it has a possible advantage of a smaller state space, because $length(Data') \leq length(Data)$. It is more precise than Steensgaard's but incomparable to Andersen's analysis. Figure 3 shows a program for which, at control point 3, Andersen's, Hind's, Steensgaard's, and the algorithm sketched above (called DF-Steen in the figure) all give different points-to relations.

7. CONCLUSION

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 existing points-to algorithms but also to devise new points-to algorithms. Section 6.2 contains an example of such an algorithm. Our program transformations can also be used outside of points-to analysis. For example, in Section 4.6 we showed how data flow analysis can be defined as the reachability analysis of a transformed program.

We plan to consider next the transformations embodied in context-sensitive versions of points-to analyses.

8. REFERENCES

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proceedings of CAV 1992*, pages 260–273, 1992.

[3] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of POPL 1992*, pages 343–354, 1992.

[4] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL '77*, pages 238–252, 1977.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[6] M. Fahndrich, J.S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of PLDI '98*, pages 85–96, 1998.

[7] Rakesh Ghiya, Daniel M. Lavery, and David C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *Proceedings of PLDI 2001*, pages 47–58, 2001.

[8] N. Heintze and O.Tardieu. Ultra-fast alias analysis using cla. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[9] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of PLDI 2001*, pages 24–34, 2001.

[10] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. on Programming Languages and Systems*, 21(4):848–894, 1999.

[11] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1–6, January 1997.

[12] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of PLDI '92*, pages 235–248, 1992.

[13] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, Fla., 1991.

[14] David McAllester. On the complexity analysis of static analyses. In *Proceedings of SAS 2001*, 2001.

[15] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of CAV 2000*, 2000.

[16] F. Nielson, H. Nielson, and C. Hankin. Principles of program analysis: Flows and effects, 1999.

[17] Thomas Reps. Program analysis via graph reachability. information and software technology. *Information and Software Technology*, 40(11-12):701–726, 1998.

[18] David Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of POPL '98*, 1998.

[19] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of POPL '97*, 1997.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of POPL '96*, 1996.

[21] Bernhard Steffen. Generating data-flow analysis algorithms for modal specification. *Science of Computer Programming*, 21:115–139, 1993.

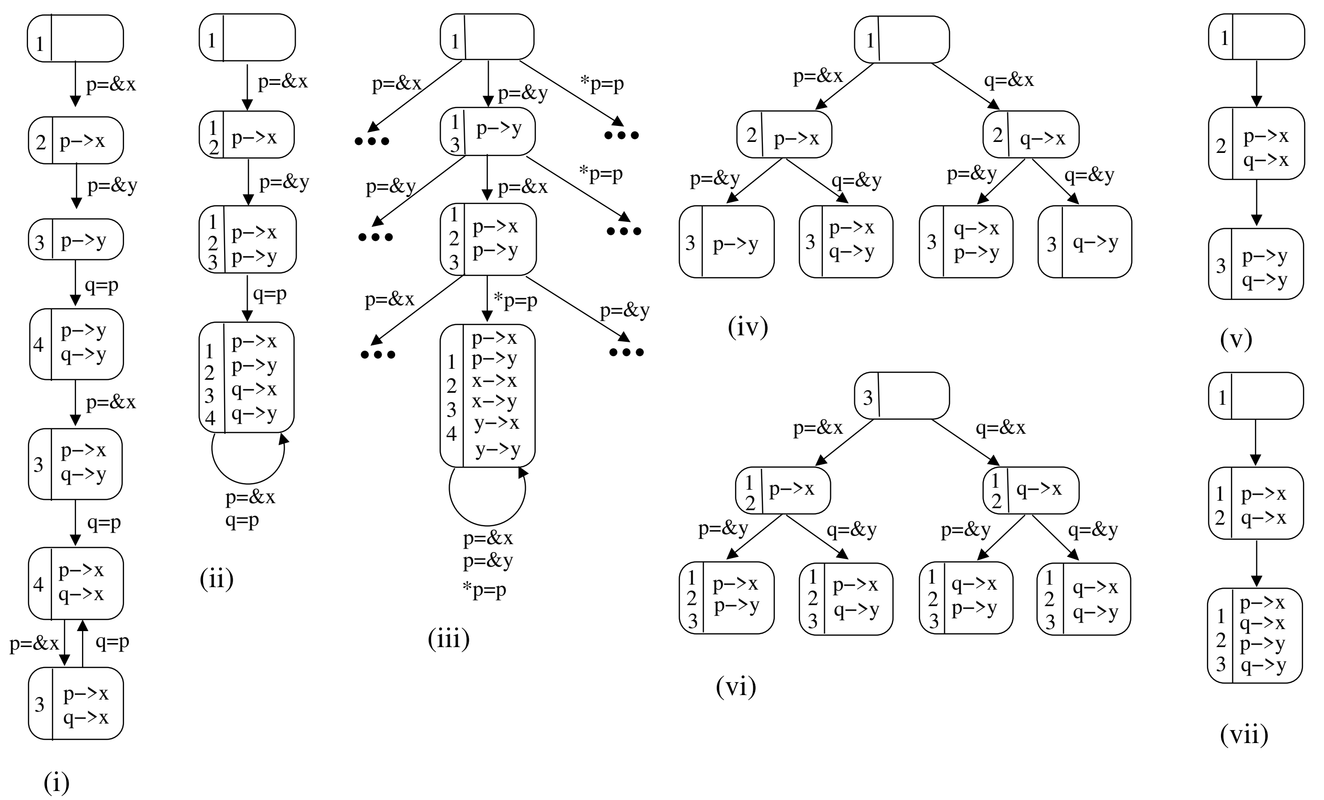

Figure 4: LTSs derived from some of the examples of Figure 2