Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints

Manuel Fähndrich Jakob Rehof Manuvir Das

Microsoft Resarch One Microsoft Way
Redmond WA, 98052
{maf,rehof,manuvir}@microsoft.com

Abstract

This paper shows that a type graph (obtained via polymorphic type inference) harbors explicit directional flow paths between functions. These flow paths arise from the instantiations of polymorphic types and correspond to call-return sequences in first-order programs. We show that flow information can be computed efficiently while considering only paths with well matched call-return sequences, even in the higher-order case. Furthermore, we present a practical algorithm for inferring type instantiation graphs and provide empirical evidence to the scalability of the presented techniques by applying them in the context of points-to analysis for C programs.

1 Introduction

Context-sensitivity based on Hindley-Milner style polymorphic type inference is in wide spread use due to its good practical running time. The low cost of such inference based analyses stems from the use of unification to model *intra*-procedural dependencies of values. *Inter*-procedural dependencies of values is captured by instantiations of polymorphic function types, but this information is generally ignored.

In this paper we study the flow of values between functions characterized by instantiations of polymorphic types. To make matters concrete, we study a polymorphic version of Steensgaard's type-based points-to analysis [Ste96]. Our analysis is flow-insensitive, i.e., statement order is ignored, but it is context-sensitive, i.e., the effects of one call site do not pollute the results at another call site to the same function. The intra-procedural part of the analysis uses undirected dependencies.

The analysis has two phases. In the first phase we build a *type instantiation graph* (TIG) using a polymorphic inference algorithm similar to ML's type reconstruction, but based on instantiation (or semi-unification) constraints [Hen93]. In the second phase, points-to information for individual program points is computed by answering simple reachability queries on the type instantiation graph. Contributions of this paper are:

- Our flow computation works without change on higher-order programs. Previous work on context-sensitive flow analysis is either restricted to first-order programs (e.g. [CRL99, LH99]), reduces the higher-order case to first-order by computing a call-graph approximation through other means (e.g. [WR99, CGS+99]), or computes a global fix-point, revisiting function descriptions as function pointers are discovered (e.g. [WL95]). Our technique enables us to compute flow information in the higher-order case directly, without the need for a call-graph approximation or an expensive global fix-point.[1]
- Our flow algorithm is efficient. Individual queries (e.g. all sources-one sink) can be answered completely on demand in linear time (in the size of the type instantiation graph). Furthermore, the algorithm is simple, since queries correspond to graph reachability questions.
- The analysis is practical and scales to large programs. The GCC SPEC benchmark is analyzed in under 3 minutes (not including parse time).
- Even though value dependencies are represented as equivalence classes intra-procedurally, flow of values across procedure boundaries is directional. Our flow computation can be retrofitted to existing analyses based on Hindley-Milner typing, extending them into flow analyses.

Many context-sensitive flow analyses based on function summaries (e.g. [CRL99, LH99, FFA00]) are presented as a two phase computation. In phase 0, information is propagated from the callees (where it originates) to the callers. In phase 1, information is propagated from callers to callees. This information represents summary information within a callee from all contexts. Our results show that this phase distinction is present on each individual flow path and generalizes to the higher-order case.

The rest of the introduction defines what we mean by context-sensitivity and flow. Section 2 defines our framework of constraint based type inference. Section 3 presents our constraint resolution algorithm and describes the construction of the type instantiation graph. Section 4 presents the flow computation on the type instantiation graph. Experimental results proving the scalability are shown in Section 5. The paper concludes with related work.

[1] In Section 1.1 we discuss different notions of context-sensitivity with distinct cost/precision trade-offs.

1.1 Context-sensitivity in higher-order programs

Context sensitivity in first-order programs is defined in terms of valid call-return paths. A path is valid if paired up call and return edges on that path are associated with the same call site. For higher-order programs (function pointers in C) we must first define what context-sensitivity means for indirect function calls. We explain the context-sensitivity of an analysis in terms of a conceptual copying of function bodies. Consider the example program below:

```
typedef int (*FIP)(int *);

int f(int *p) {...}
int g(int *q) {...}

void foo(int a, int b, int c) {
  int ra, rb;
  FIP fp = c?f_i:g_j;

(1)  ra = fp_1(&a);
(2)  rb = fp_2(&b);
}
```

Function pointer `fp` is assigned either function `f` at occurrence i or function `g` at occurrence j. Indirect call sites 1 and 2 are using `fp`. Consider a polymorphic analysis of this program that treats each indirect call to a particular function independently from other calls. Such an analysis corresponds to analyzing the expanded program shown in Figure 1 monomorphically (not context-sensitive). In this expansion, we have two copies $\mathtt{f}_{i1}$ and $\mathtt{f}_{i2}$ of function `f`, one per indirect call site, and similarly for function `g`. This context-sensitivity is based on functions reaching individual call-sites. It is expensive, since the number of instances of a function depends on the number of indirect call sites it flows to.

Another form of context-sensitivity for higher-order programs is adopted in this paper. We only allow one copy of a function per occurrence of a function symbol. In our example, this form corresponds to analyzing the expanded program in Figure 2 monomorphically. There is one copy of function `f` corresponding to occurrence i and one copy of function `g` corresponding to occurrence j. The same function is called at the two indirect call-sites 1 and 2. This form has the advantage that it generates only as many instances of a function `f` as there are occurrences of the symbol `f` in the program. On the other hand, the use of fewer instances may lead to less precise results. This form of context-sensitivity corresponds to recursive let-polymorphism [Myc84]. Note that in the first-order case, the two approaches are identical, since function symbols only occur at call sites.

The analogy of copying functions is only conceptual (and obviously does not apply in the recursive case). In practice, we analyze a function only once, producing a compact summary (its polymorphic type). At each instance, a copy of this summary is used.

1.2 Flow information

We now define the flow queries we compute. Assume that each sub-expression in a program is annotated by a label ℓ. We ask questions such as "Do values arising at label ℓ_1 in the program flow to a program point labelled ℓ_2?" More precisely, we associate the label ℓ of an expression e with the type τ of e. The queries are then answered by tracing paths on the *type instantiation graph* (TIG) instead of the program. The TIG embodies the type relations necessary for the program to be well-typed.

2 Constraint based type inference

Constraint based type inference automatically infers types for a program by generating a set of type constraints from the program text and then solving the constraints. The following section introduces types and constraints, and in the next section we give an algorithm for solving these constraints. Note that the types we infer do not necessarily coincide with standard C types.

2.1 Types and locations

Type based flow analysis assigns *types* and *locations* to program objects.[2] Type expressions, ranged over by τ, are built from variables, ranged over by α, and type constructors:

$$\tau \quad ::= \quad \alpha \mid (\tau_1, \ldots, \tau_n) \rightarrow^{\ell} \tau \mid ptr^{\ell}(\tau)$$

In order to capture flow properties, type constructors ($\rightarrow$, ptr) are labeled with *flow variables*, ranged over by ℓ. The type $(\tau_1, \ldots, \tau_n) \rightarrow^{\ell} \tau$ is the type of functions mapping arguments of types $\tau_1, \ldots, \tau_n$ to a result of type τ; the type $ptr^{\ell}(\tau)$ is the type of pointers pointing to objects of type τ. Flow variables, ℓ, are used to uniquely name (types of) program objects of interest, such as pointers, functions and locations. For example, $ptr^{\ell}(\tau)$ is a pointer to the location named ℓ. In the type $(\tau_1, \ldots, \tau_n) \rightarrow^{\ell} \tau$, we can think of ℓ as the address of a particular function.

In order to model the distinction between L-values and R-values in C, we introduce *locations* of the form $[\tau]^{\ell}$, denoting a memory location named ℓ, holding values of type τ. Locations $[\tau]^{\ell}$ are associated with L-values, whereas types τ are associated with corresponding R-values. Consider the following C program fragment as a simple example:

```
int x, *p;
p = &x;
```

In this program, symbol `x` is associated with a location $[\alpha]^{\ell}$, and `p` is associated with $[ptr^{\ell}(\alpha)]^{\ell'}$. The types are interpreted as saying that after executing the assignment, `p` points to location ℓ associated with `x`, *i.e.*, `p` points to `x`.

2.2 Constraint generation

Figure 3 gives a representative set of constraint generation rules for our pointer analysis of C programs. The rules for expressions use typing judgments of the form $A \vdash e : \sigma / C$, where σ is either a location or a type. The meaning of such a judgment is that in *type environment* A expression e can be given type or location σ, on condition of the *constraint set* C. A type environment A is a set of assignments of the form $x : [\tau]^{\ell}$, assigning location $[\tau]^{\ell}$ to program variable x. A constraint set C is a finite set of simultaneous equalities and inequalities between types, written as $\tau = \tau'$ and $\tau \leq_p^i \tau'$, respectively. An equality $\tau = \tau'$ means that the types τ and τ' must be unified. Inequalities are called *instantiation constraints*. We now explain the meaning of inequalities together with some of the the rules of Figure 3.

[2] We should emphasize that the techniques presented in this paper generalize to any conventional type language. The one chosen here just makes it easy for us to show how our techniques apply to the analysis of C programs.

```
int f_i1(int *p) {...}
int f_i2(int *p) {...}

int g_j1(int *q) {...}
int g_j2(int *q) {...}

void foo(int a, int b, int c) {
  int ra, rb;

  if (c) {
    ra = f_i1(&a);
    rb = f_i2(&b);
  }
  else {
    ra = g_j1(&a);
    rb = g_j2(&b);
  }
}
```

Figure 1: One expansion per function per call-site

```
int f_i(int *p) {...}
int g_j(int *q) {...}

void foo(int a, int b, int c) {
  int ra, rb;

  if (c) {
    ra = f_i(&a);
    rb = f_i(&b);
  }
  else {
    ra = g_j(&a);
    rb = g_j(&b);
  }
}
```

Figure 2: One expansion per function occurrence

2.2.1 Instantiation constraints

Polymorphism [Mil78, DM82] specializes the type of a function at each use by instantiating the type. Instantiation is usually expressed by substitution of bound variables. Instead we use inequalities of the form $\tau \leq_p^i \tau'$ to express that τ' is an instance of τ [Hen93, KTU94]. An inequality $\tau \leq_p^i \tau'$ requires that τ' must be a *substitution instance* of τ, *i.e.*, $\tau' = R(\tau)$ for some substitution R (mapping of type variables to types).

Constraints of the form $\tau_{def} \leq_p^i \tau_{use}$ are generated whenever rule [Fun] in Figure 3 is applied. Here τ_{def} represents the type inferred from the definition of a function f (via rule [Def]), whereas τ_{use} represents the instance type inferred for a particular use of f (for example via rule [Call]). A formal definition of the solution to a set of constraints is given in Section 3. Disregarding for the moment the indices i and p on the inequality symbol [3], we briefly outline how functions get typed using the rules of Figure 3 by a simple example.

2.2.2 An example

Consider the identity function `id` defined by

```
id(x){ return x; }
```

This function can be given the polymorphic type $\alpha \to^\ell \alpha$, where the type variable α may be instantiated to any type at uses of `id`. The definitional type $\alpha \to^\ell \alpha$ is inferred by rule [Def]. Assuming location $[\alpha]^{\ell_1}$ for `x`, we conclude by rule [Ret], that the return type of `id` (denoted $\alpha_{ret(\mathtt{id})}$ in the rule) is α. By rule [Def] we then conclude that the definitional type of `id` (denoted $\alpha_{\mathtt{id}}$ in the rule) is $\alpha \to^\ell \alpha$. Now, suppose we apply `id` to a pointer `p` of type $ptr^{\ell'}(\gamma)$ at a call site `id(p)`. Because this call site mentions the function name `id`, the rule [Fun] gives rise to a constraint of the form $\alpha_{\mathtt{id}} \leq_p^i \beta$, where β is a fresh variable for the type of `id` at this call site. Since we already found that $\alpha_{\mathtt{id}} = \alpha \to^\ell \alpha$, the instantiation constraint is equivalent to $\alpha \to^\ell \alpha \leq_p^i \beta$.

[3] The meaning of index i is explained in Section 2.2.3, and the meaning of the index p is explained in Section 2.2.4.

Finally, rule [Call] requires the type of the argument `p` to be equal to the domain type of `id`, so the type of `id` at this call site (*i.e.*, β), must have the form $ptr^{\ell'}(\gamma) \to^{\ell''} \tau$, for some type τ. The inequality above therefore becomes equivalent to

$$\alpha \to^\ell \alpha \leq_p^i ptr^{\ell'}(\gamma) \to^{\ell''} \tau \qquad (1)$$

By choosing τ to be $ptr^{\ell'}(\gamma)$, the inequality (1) can be solved by instantiating α to $ptr^{\ell'}(\gamma)$.

Notice that the order in which we process the definition and the uses of a function symbol does not matter. If we process a use before a definition, we can still express the fact that the definitional type must instantiate to the use type, because we collect the constraint $\alpha_f \leq_p^i \tau_{use}$ at the use site. When the definition of f is processed, an equality $\alpha_f = \tau_{def}$ is generated, leading to $\tau_{def} \leq_p^i \tau_{use}$. Instantiation constraints therefore allow type inference to be performed in a fully modular way [OJ97].

2.2.3 Constraint indices

Textual references to a function symbol f in a program are assumed to be tagged with a unique index i identifying the occurrence, written f_i. For each occurrence f_i, rule [Fun] gives rise to a constraint $\alpha_f \leq_p^i \beta$, where α_f is a placeholder for the definitional type of f and β is a placeholder for the instance type required at the particular use f_i. The index i of the occurrence is attached to the corresponding inequality and is used by the constraint solver to keep track of inequalities inferred from this one.

2.2.4 Constraint polarities

Inequalities $\leq_p^i$ are further annotated by a *polarity* p. Polarities are used by our algorithm to direct the flow computation. Intuitively, polarities keep track of input and output positions in types, and they do so in a way that works for higher order programs. Formally, polarities are elements in the set $\{+, \div, \top\}$, which is ordered by $+ \leq \top$, $\div \leq \top$. The polarity $+$ represents positive polarity, $\div$ negative polarity, and $\top$ both positive and negative polarity.

Expressions

$$[\text{ Fun }]\frac{\beta \text{ fresh}}{A \vdash f_i : \beta / \{\alpha_f \leq^i_+ \beta\}}$$

$$[\text{ Var }]\frac{A(x) = [\tau]^\ell}{A \vdash x : [\tau]^\ell / \emptyset}$$

$$[\text{ Call }]\frac{\begin{array}{c} A \vdash e_0 : \tau_0 / C_0 \\ A \vdash e_i : \tau_i / C_i \ (i = 1 \ldots n) \\ C' = \cup_{j=0}^{n} C_j \\ C'' = \{\tau_0 = (\tau_1, \ldots, \tau_n) \to \tau\} \end{array}}{A \vdash e_0(e_1, \ldots, e_n) : \tau / C' \cup C''}$$

$$[\text{ Assign }]\frac{\begin{array}{c} A \vdash e_1 : [\tau]^\ell / C_1 \\ A \vdash e_2 : \tau', / C_2 \\ C_3 = \{\tau = \tau'\} \end{array}}{A \vdash e_1 \texttt{ = } e_2 : \tau' / C_1 \cup C_2 \cup C_3}$$

$$[\text{ Rval }]\frac{A \vdash e : [\tau]^\ell / C}{A \vdash e : \tau / C}$$

$$[\text{ Addr }]\frac{A \vdash e : [\tau]^\ell / C}{A \vdash \&e : ptr^\ell(\tau) / C}$$

$$[\text{ Deref }]\frac{A \vdash e : ptr^\ell(\tau) / C}{A \vdash *e : [\tau]^\ell / C}$$

Statements

$$[\text{ Sequence }]\frac{\begin{array}{c} A \vdash s_1 : C_1 \\ A \vdash s_2 : C_2 \end{array}}{A \vdash s_1 \texttt{;} s_2 : C_1 \cup C_2}$$

$$[\text{ Local }]\frac{A, x : [\tau]^\ell \vdash s : C}{A \vdash \texttt{local } x \texttt{ in } s : C}$$

$$[\text{ Def }]\frac{\begin{array}{c} A, x_1 : [\tau_1]^{\ell_1}, \ldots, x_n : [\tau_n]^{\ell_n} \vdash s : C \\ C' = \{\alpha_f = (\tau_1, \ldots, \tau_n) \to^\ell \alpha_{ret(f)}\} \end{array}}{A \vdash f(x_1, \ldots, x_n)\{s\} : C \cup C'}$$

$$[\text{ Ret }]\frac{\begin{array}{c} A \vdash e : \tau / C \\ C' = C \cup \{\alpha_{ret(f)} = \tau\} \end{array}}{A \vdash \texttt{return}_f e : C'}$$

Figure 3: Constraint generation for analysis of C

We say that a term τ occurs positively (resp. negatively) in a type expression τ', if τ occurs nested to the left of the function type constructor ($\to$) in τ' an even (resp. uneven) number of times. For example, in the type expression $(\alpha \to \beta) \to \gamma$, the type $\alpha \to \beta$ occurs negatively, α and γ occur positively, and β occurs negatively. Targets τ of pointer types $ptr(\tau)$ occur at polarity $\top$.

The notion of polarities is standard in type theory. We transfer this notion to inequalities in the following way. Initially, every inequality generated by rule [Fun] in Figure 3 has positive polarity (*i.e.*, all inequalities have the form $\leq^i_+$). During constraint resolution inequalities *propagate* to their subterms, where polarities switch according to the polarities of the subterms.

Our flow computation in Section 4 interprets the instantiation relations as flow-edges. Intuitively, the derived constraint $\alpha \leq^i_\div ptr^{\ell'}(\gamma)$ can be interpreted as implying that (the type of) the actual argument `p` ($ptr^\ell(\gamma)$) flows to (the type of) the formal parameter `x` (α), and the constraint $\alpha \leq^i_+ \tau$ as implying that (the type of) the return value of `id` (α) flows to (the type of) the return point of the call (τ). In this interpretation, the direction of flow is governed by polarities: the flow moves *opposite* the direction of the *negative* inequality ($\leq^i_\div$) but *along* the direction of the *positive* inequality ($\leq^i_+$).

3 Constraint resolution

In this section we first give a technical definition of what it means to solve a set of constraints. We then go on to present our constraint resolution algorithm.

3.1 Semi-unification

The problem of solving constraint systems involving equalities and inequalities of the form described above is well-known and usually referred to as the *semi-unification* problem [Hen93, KTU94].

Following is a technical definition of what it means to solve such systems (the reader may wish to skip it on a first reading; more background can be found in [Hen93, KTU94]).

Definition 3.1 A *substitution* is a function from type variables to types. An *inequality* $\tau \leq^i_p \tau'$ is called *solvable* if τ' is a substitution instance of τ, *i.e.*, one has $R_i(\tau) = \tau'$ for some substitution R_i.

A substitution S is a *solution to a constraint set* C (consisting of simultaneous equalities and inequalities) iff one has $S(\tau) = S(\tau')$ for every equality $\tau = \tau' \in C$ and $S(\tau) \leq^i_p S(\tau')$ is solvable (by some substitution R_i) for every inequality $\tau \leq^i_p \tau' \in C$. Notice that for each particular index i, the set of inequalities associated with i ($S(\tau) \leq^i_p S(\tau')$) may be solved by different substitutions R_i. □

Semi-unification is known to be undecidable [KTU93], but a practical semi-decision procedure was defined by Henglein [Hen93]. So far, no natural counterexample (*i.e.*, a program that would make the algorithm loop) is known, and the results of the present paper corroborate the practicality of the algorithm.

3.2 Algorithm

The core of our constraint resolution algorithm is shown in Figure 4. It uses auxiliary operations defined in Figure 5.

1. *Input* : A finite set C of constraints, of the form $t \leq_p^i t'$ or $t = t'$.
2. *Initially* : $Wlist := C$
3. *Iteration* :

while $Wlist \neq \emptyset$ **do**

$\tau\ op\ \tau' :=$ FETCH($Wlist$);
$L :=$ FIND(τ);
$R :=$ FIND(τ');
if $L = R$ **then continue**;
if L **matches** $c^\ell(\tau_1, \ldots, \tau_n)$ **and** R **matches** $d^{\ell'}(\tau'_1, \ldots, \tau'_n)$ **and** $c \neq d$ **then fail**;

switch (op)
case $=$
UNION(L, R);

if L **matches** $c^\ell(\tau_1, \ldots, \tau_n)$ **and** R **matches** $c^{\ell'}(\tau'_1, \ldots, \tau'_n)$
then
$Wlist := Wlist \cup \{\ell = \ell'\} \cup \{\tau_k = \tau'_k \mid k = 1 \ldots n\}$;

case $\leq_p^i$
$storable :=$ STORE$(L \leq_p^i R)$;
if not $storable$ **then continue**;

if L **matches** $c^\ell(\tau_1, \ldots, \tau_n)$ **and** R **matches** $c^{\ell'}(\tau'_1, \ldots, \tau'_n)$
then
$Wlist := Wlist \cup \{\ell \leq_p^i \ell'\} \cup \{\tau_k \leq_{p_k}^i \tau'_k \mid p_k =$ PROPAGATE$(c, p, k), k = 1 \ldots n\}$;

if L **matches** $c^\ell(\tau_1, \ldots, \tau_n)$ **and** R **matches** α
then
if EOC(α, L)
then
$Wlist := Wlist \cup \{\alpha = L\}$;
else
$Wlist := Wlist \cup \{\alpha = c^{\ell'}(\beta_1, \ldots, \beta_n), L \leq_p^i \alpha\}$
where $\ell', \beta_1, \ldots, \beta_n$ **are fresh**;
end; (* while *)

(Notation: τ **matches** τ' is syntactic pattern matching.)

Figure 4: Algorithm $\mathcal{I}$ for solving instantiation constraints

Our algorithm extends Henglein's algorithm in [Hen93] in two directions. First, it allows recursive types (so that type equations such as $\alpha = \alpha \to \alpha$ have solutions) via cyclic unification ([ASU88] Section 6.6), and, secondly, our procedure propagates polarities. While Henglein's algorithm was specified as an abstract graph rewriting system in [Hen93], we give a more concrete, worklist based algorithm. We do so for three reasons. First, the algorithm builds the instantiation graph over which the flow computation takes place, and we attempt to give enough detail that the reader can see how our flow computation is supported by the constraint solver in practice. Second, we wish to demonstrate the scalability of the particular implementation strategy chosen here. Third, handling recursive types turns out to be challenging, and we hope that our specification can serve as an off-the-shelf, easily implementable solution.

Since the main structure of the algorithm follows [Hen93], we will focus on the extensions, polarities, recursive types and the construction of type instantiation graphs, which are concentrated in the operations PROPAGATE, EOC and STORE.

3.2.1 Type representation and recursive types

In Figure 4, non-variable types are written as $c^\ell(\tau_1, \ldots, \tau_n)$ with c and d ranging over type constructors $(\to, ptr)$. Types are represented as possibly cyclic graphs (cycles represent recursive types), whose nodes represent type constructors and type variables, as in [ASU88] Section 6.6. A single type constructor, say c, can be represented by many different nodes, corresponding to different occurrences of the constructor c.

Unification of types is implemented via equivalence relations based on fast UNION/FIND, and nodes are instrumented with UNION/FIND-information in a standard manner ([ASU88] Section 6.6).

Performing a UNION operation on two constructed types $c^\ell(\tau_1, \ldots, \tau_n)$ and $c^{\ell'}(\tau'_1, \ldots, \tau'_n)$ involves choosing one of the nodes representing the main constructor c of one of the

FETCH(*Wlist*) =
  if *equalities*(*Wlist*) $\neq \emptyset$
  then return POP(*equalities*(*Wlist*));
  else return POP(*inequalities*(*Wlist*));

UNION(τ, τ') =
  if τ **is a variable**
  then $rep := \tau'$; $other := \tau$;
  else $rep := \tau$; $other := \tau'$;
  $ecr(other) := rep$;
  $Wlist := Wlist \cup \{rep \leq^i_p \tau \mid other \mapsto^i_p \tau\}$;

STORE($\tau \leq^i_p \tau'$) =
  if $target(\tau, i)$ **is undefined**
  then
    $target(\tau, i) := \tau'$;
    $polarity(\tau, i, \tau') := p$;
    return true;
  else
    if $\tau' \neq target(\tau, i)$
    then
      $polarity(\tau, i, \tau') := polarity(\tau, i, \tau') \sqcup p$;
      $Wlist := Wlist \cup \{\tau' = target(\tau, i)\}$;
    return false;

PROPAGATE(c, p, k) =
  if c **is co-variant in** k
  then return p;

  if c **is contra-variant in** k
  then return $\neg p$;

  if c **is invariant in** k
  then return $\top$;

EOC(τ, τ') =
  if $\exists \tau_1, \ldots, \tau_n$ **such that**
    $\tau' = \tau_1$ **and** $\tau_1 \mapsto \tau_2 \mapsto \ldots \mapsto \tau_n = \tau$
    and τ **is proper subterm of** τ'
  then return *true*;
  else return *false*;

  (Notation: $\tau \mapsto \tau'$ iff $\exists i, p.\ \tau \mapsto^i_p \tau'$.)

Figure 5: Auxiliary procedures for Algorithm $\mathcal{I}$

two types as equivalence class representative (in this case, either the node for c labeled with ℓ or the node for c labeled with ℓ' is chosen as representative). Labels on type constructors respect node equivalence such that label ℓ is equivalent to label ℓ' if and only if the constructor nodes corresponding to c^ℓ and $c^{\ell'}$ are equivalent under UNION/FIND. Labels on type constructors can be used as identifiers of (equivalence classes of) nodes, so that c^ℓ denotes a particular (equivalence class) node representing constructor c.

3.2.2 Main loop

The main loop of the algorithm (Figure 4) uses a worklist, which always holds the remaining unsolved constraints (initially the input constraint set C). In the loop, an unsolved constraint $\tau\ op\ \tau'$ is popped (using the operation FETCH) from the worklist, where *op* is either an equality symbol (=) or an inequality ($\leq^i_p$). The FETCH operation chooses equalities before inequalities. This scheme identifies as many nodes in the type structure as early as possible, leading to improved performance. If the terms τ and τ' of the constraint have distinct root constructors c and d, the constraint cannot be solved and the algorithm fails.[4] Otherwise, the algorithm branches on the form of the relation (*op*) of the constraint. If *op* is equality, =, a UNION is performed, and equalities are propagated downwards on the type trees.

If *op* is inequality, $\tau \leq^i_p \tau'$, a STORE operation is performed. This operation *caches* instantiations and applies the following constraint closure rule:

$$\tau \leq^i_p \tau_1 \quad \wedge \quad \tau \leq^i_{p'} \tau_2 \quad \Rightarrow \quad \tau_1 = \tau_2 \tag{2}$$

Notice that, in this rule, the index i must be the same on both inequalities. The rule ensures that any two occurrences of the same variable get instantiated to the same type, within a single instantiation. For example, the constraint (1) from our running example in Section 2.2.2, $\alpha \to^\ell \alpha \leq^i_+ ptr^{\ell'}(\gamma) \to^{\ell''} \tau$, implies $\alpha \leq^i_\div ptr^{\ell'}(\gamma)$ and $\alpha \leq^i_+ \tau$, as explained earlier. Hence, we must have $ptr^{\ell'}(\gamma) = \tau$.

For an inequality $\tau \leq^i_p \tau'$, the STORE operation stores a reference to the node representing the main constructor of τ' at the node representing the main constructor of τ. This reference also carries the information contained in the index i and polarity p. The algorithm uses the notation $\tau \mapsto^i_p \tau'$ for cached instantiations as well as $target(\tau, i) = \tau'$ and $polarity(\tau, i, \tau') = p$. If a node has already been stored at τ and index i (*i.e.*, if $target(\tau, i)$ is defined), the STORE operation emits the equation $\tau' = target(\tau, i)$, thereby implementing rule (2).

Operation STORE defines the *type instantiation* graph on which our flow computations will be performed (see Section 3.3). The instantiation graph is further used by the UNION operation. Suppose we have a constraint of the form $\tau = \tau'$, and that τ is chosen as equivalence class representative. Then the type τ' will effectively be "killed". If $target(\tau', i)$ is defined (for any i), we must make sure that targets of τ' do not get lost as τ' is killed; this is done by substituting τ for τ', and we therefore emit $\tau \leq^i_p \tau''$ to the worklist for all targets τ'' of τ'; these will then become targets of τ by future STORE operations, as described above.

The operation PROPAGATE gets called when constraints are propagated (added to the worklist), as explained in Section 2.2.4.. This operation finds the appropriate polarity for the propagated constraints.

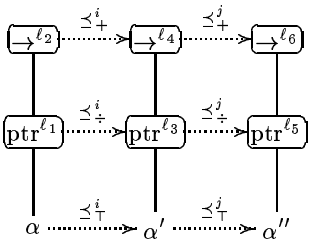
The *extended occurs check* [Hen93], implemented in the operation EOC, ensures that cyclic instantiation constraints of the form $\tau' \preceq^{i_1}_{p_1} \cdots \preceq^{i_n}_{p_n} \tau$ where τ is a proper subterm of τ' get transformed into equalities, thereby blocking cases where infinite instantiations might otherwise occur. The check EOC(τ, τ') is implemented by a depth-first search over the instantiation graph starting from the root node of τ'. If the root node of τ is reachable via instances from the root node of τ', one checks whether the root of τ properly occurs within τ'. The occurs-check is done by a depth-first search of the term-graph representing τ'.

3.3 Type instantiation graph

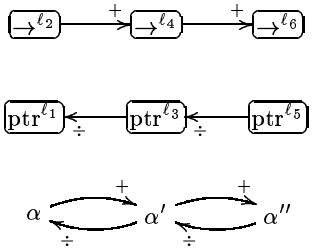
The instantiation cache maintained by STORE defines the type instantiation graph and represents a *complete trace*

[4] For C, we use finite sum types as in [Ste96], guaranteeing that constraints solving never fails.

```
        typedef void (*FIP)(int *);

        void f(int *p) {        /* f : ptr^ℓ1(α) →^ℓ2 void */
(1)       ... *p ...;
        }

        FIP g() {               /* g : void →^ℓ7 (ptr^ℓ3(α') →^ℓ4 void) */
(i)       return(&f);           /* f_i : ptr^ℓ3(α') →^ℓ4 void */
        }

        h() {
          int c;                /* c : [α'']^ℓ5 */
(j)       FIP fp = g();         /* g_j : void →^ℓ8 (ptr^ℓ5(α'') →^ℓ6 void) */

(2)       fp(&c);               /* fp : ptr^ℓ5(α'') →^ℓ6 void */
        }
```



Type instantiation graph



Flow graph

Figure 6: Example with returned function pointer

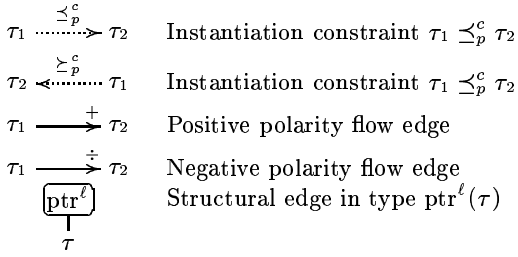


Figure 7: Edge conventions

of all instantiations during resolution. More precisely, the nodes of the graph are subterms of types and there is an edge $\tau \overset{\preceq^i_p}{\cdots\cdots\gg} \tau'$ if and only if $\tau \mapsto^i_p \tau'$, *i.e.*, the constraint resolution algorithm inferred $\tau \preceq^i_p \tau'$.

4 Computing global flow information

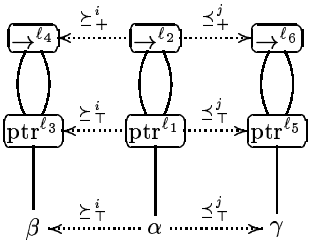
This section shows how to compute global flow information (points-to information in our running example) from the type-instantiation graph by interpreting type instantiation graphs as flow graphs. When presenting graphs, we use the conventions shown in Figure 7. Instantiations are represented as dotted edges labelled by the instantiation constraint. Their direction represents the instantiation direction (from generic to instance). Flow edges represent the actual flow direction of values and are labelled by a polarity (explained later). Nodes are particular subexpressions of types. We draw only the top-level nodes of such types, possibly surrounded by a box to help identify them as nodes. Where appropriate, we connect types with their immediate subexpressions via undirected solid edges.

We continue by examining a simple example that suggests how to interpret a type instantiation graph as a flow graph. Our examples explicitly contain function pointers to show the workings of the flow computation in the higher-order case. More details on the correctness of the approach can be found in a technical report [FRD00].

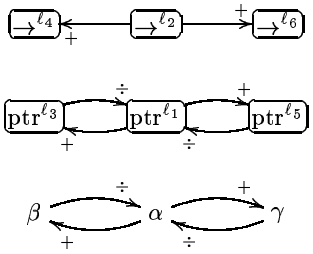
Figure 6 shows a C program with the inferred types given as comments. The instantiation graph arising from points i and j is given at the bottom left of the Figure (void nodes are not shown). Recall that labels can be thought of as labelling type nodes. We will take this view in the following explanation. Let us first ask the following query at line (2): What are the functions that are applied at this indirect call? The function node of the indirect call is $\boxed{\to^{\ell_6}}$. The C semantics tells us that function `f` identified by $\boxed{\to^{\ell_2}}$ flows to the indirect call. In our instantiation graph, node $\boxed{\to^{\ell_2}}$ is connected to $\boxed{\to^{\ell_6}}$ via the instantiation edges

These instantiations with positive polarity stem from the use of the [FUN] rule on occurrences of $\mathtt{f}_i$ and $\mathtt{g}_j$. If we view these instantiation edges as directed flow edges (as shown in Figure 6 bottom right), we can conclude that function

```
(0)     int *id(int *p) {        /* id : ptr^ℓ1(α) →^ℓ2 ptr^ℓ1(α) */
(1)       *p; return p;
        }

          foo() {
            int b;               /* b : [β]^ℓ3 */
(i)         id(&b);              /* id_i : ptr^ℓ3(β) →^ℓ4 ptr^ℓ3(β) */
          }

          bar() {
            int c;               /* c : [γ]^ℓ5 */
(j)         id(&c);              /* id_j : ptr^ℓ5(γ) →^ℓ6 ptr^ℓ5(γ) */
          }
```



Type instantiation graph



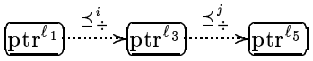
Flow graph

Figure 8: Identity example

`f` (represented by node $\boxed{\rightarrow^{\ell_2}}$) flows to the indirect function pointer `fp` (represented by node $\boxed{\rightarrow^{\ell_6}}$). We capture this observation in our first flow rule.

Rule 4.1 *Instantiation edges with positive polarity (+) translate into flow edges with the same direction. These flow edges inherit polarity +.*

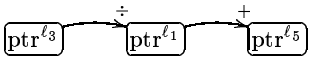
Next, we ask the flow query: what pointer value can be dereferenced by `p` at point (1)? The C semantics tells us that the address (or location) of `c` flows to `p` where it is dereferenced. The contents of `p` is uniquely identified by node $\boxed{\text{ptr}^{\ell_1}}$, and the location of `c` is labelled by $\boxed{\text{ptr}^{\ell_5}}$. Inspecting our instantiation edges in Figure 6, we notice that these two nodes are connected by the instantiation edges



where the constraints have negative polarity. In order to deduce flow from $\boxed{\text{ptr}^{\ell_5}}$ to $\boxed{\text{ptr}^{\ell_1}}$, we must treat instantiation edges with polarity $\div$ as flow edges in the *opposite* direction of the instantiation, suggesting the next flow rule:

Rule 4.2 *Instantiation edges with negative polarity ($\div$) translate into flow edges having opposite direction. These flow edges inherit polarity $\div$.*

The two flow rules yield a sound flow representation of a program described by a type instantiation graph, i.e., they completely capture all value flow in the program. But unfortunately, applying them blindly to a type instantiation graph results in a complete loss of context-sensitivity as shown by the next example in Figure 8. If we ask the question: what pointers are returned by the application of `id` at (j), we obtain the answer: pointers to locations ℓ_3 (`b`) and ℓ_5 (`c`) through the path

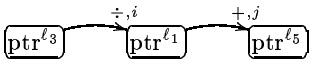


The flow of ℓ_3 (`b`) is imprecise since it traces a path from call site (i), through `id`, returning to call site (j). In order to obtain more precise answers to flow queries, we need to restrict the set of paths we consider in the reachability question, without sacrificing soundness.

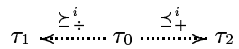
4.1 PosNeg-flow

So far we have used the polarity on instantiations to give direction to flow edges. But we have ignored the polarity of the flow edges themselves. Flow edges inherit their polarity from the instantiation edge that gives rise to the flow edge. The imprecise flow path from $\boxed{\text{ptr}^{\ell_3}}$ to $\boxed{\text{ptr}^{\ell_5}}$ above consists of a negative ($\div$) flow edge, followed by a positive (+) flow edge. In the first-order case, a negative polarity edge represents the flow edge from an actual to a formal parameter, and a positive polarity edge represents the flow of a result back to the caller. In this light, a path fragment consisting of a negative ($\div$) flow edge followed by a positive (+) flow edge represents a call/return flow. In the context-sensitive setting, such paths need only be considered if the call site of the call edge matches the site of the return edge. In our formulation, call sites are represented by indices on the instantiation edges but we chose to elide the index on flow edges for reasons addressed below. Suppose for the instant that we nevertheless attach the index of an instantiation edge when we translate it into a flow edge. Annotated with

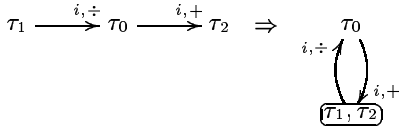
indices, the imprecise path above has the form



indicating that the call edge originates from site i, but returns to site j. Reps et. al. show in [RHS95] for the first-order case that restricting flow paths to matching call/returns is equivalent to a context-free language reachability query. In our case however, the problem is simpler since our type inference explained in the previous section collapses matching sequences of a negative edge ($\div$) followed by a positive ($+$) edge by Rule (2). In other words, presented with two instantiation edges with matching indices i



the constraint resolution produces an equality $\tau_1 = \tau_2$, effectively collapsing nodes τ_1 and τ_2. On the equivalent flow graph the collapse amounts to



Thus, flow sequences representing matching call/returns are already collapsed in the type instantiation graph and can be ignored in the flow computation. We thus disregard all paths containing a negative edge followed by a positive edge in our graph reachability problem. Thus, the only valid flow paths consist of any number of positive edges, followed by any number of negative edges. We call such paths *PosNeg-paths.*

Definition 4.1 [PosNeg-Flow] We say that there is flow from a node labelled ℓ_1 to a node labelled ℓ_2 in a flow graph G constructed via Rules 4.1 and 4.2, if there exists a PosNeg-path in G from ℓ_1 to ℓ_2. □

The soundness of our PosNeg-flow in the higher-order case relies on the fact that each instantiation constraint produced during constraint generation can be interpreted as a subtyping constraint with the obvious flow interpretation. A detailed discussion of the mechanism and a proof of its soundness can be found in [FRD00].

4.2 Complexity

Our flow formulation answers individual queries (e.g. all source-one sink) in linear time in the size of the type instantiation graph. All queries can be answered in quadratic time. The complexity is directly related to the fact that the end-points of matching call/return edges are collapsed.

Restricting each individual query to PosNeg-paths leads to a simple algorithm that is naturally demand-driven. In contrast, two phase algorithms require that phase 0 (up propagation) be entirely completed before any propagation of phase 1 (down propagation), otherwise context-sensitivity is lost. Implementing a completely demand-driven version of these two-phase algorithms is thus challenging.

In [FRD00], we study the generalization of the present flow analysis to directional edges. In the generalized case, flow queries are answered via context-free language reachability as described by Reps et. al [RHS95, MR97]. Our technical report contains a cubic algorithm for answering all or any single query.

5 Experiments

This section shows that our techniques scale to large programs by presenting numbers for an implementation of the described type inference and flow algorithms. We show the precision improvements gained over a monomorphic version in the context of points-to analysis. The monomorphic analysis is a version of Steensgaard's points-to analysis [Ste96].[5]

We analyze a range of C programs from the SPEC benchmark suite. The raw numbers are given in Table 1. All experiments were run on a Dell Precision 610 with 512MB of memory. To measure the precision of the points-to analysis, we count points-to set sizes at static pointer dereference points only (direct accesses to arrays are not counted as dereferences).

Types of global variables are treated monomorphically. Each occurrence of `malloc` generates a fresh global variable representing the class of heap cells arising at that point. Our implementation uses sum types to represent C values that can be either pointers or functions.

Figure 9 shows the reductions in the average points-to set size obtained through polymorphism. The most dramatic reduction is obtained for Vortex, where the average points-to set size drops from 1661 to 62. Even for GCC, we get almost a factor of 5 reduction in the average points-to set size.

Figure 10 gives the running time of the monomorphic and the polymorphic analyses. We give the time per abstract syntax tree node to show the scaling behavior. The running time is broken down into monomorphic running time, time for computing the polymorphic type instantiation graph (in excess of the monomorphic time), and the time to compute the flow result. The numbers show that the polymorphic type instantiation graph can be computed with little overhead over a monomorphic analysis. The time to compute the flow information however is a substantial fraction of the analysis time. Fortunately, the absolute times are still small (<3 minutes for gcc). The flow computation is currently implemented as a non-demand driven, forward-only flow, where each symbol is propagated along all PosNeg-paths. We believe this naive implementation can be improved substantially.

Finally, Figure 11 shows the space consumption of the polymorphic analysis as a factor of the space consumption of the monomorphic analysis. The space is broken down into type nodes and instantiation edges. The space overhead of polymorphism is substantial and currently the main inhibitor to scaling the analysis to very large programs. We are able to construct the final type instantiation graph for MS Word (2.1MLoc) within 512MB of memory, but exceed memory during the flow computation. Finding ways to further reduce the memory consumption is part of future work.

6 Related work

Jagannathan and Wright [JW95] and Nielson and Nielson [NN97] study flow analysis frameworks. These frameworks contain analyses that can distinguish between a number of distinct contexts in which functions are used. They differ from our technique in that functions are reanalyzed in each new context.

[5] Library function stubs are treated polymorphically, no conditional unification is used.

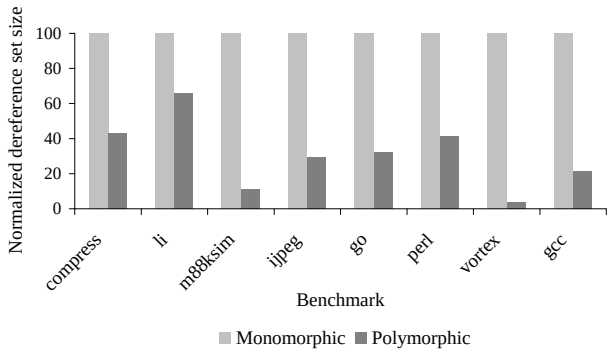


Figure 9: Reductions of points-to sizes

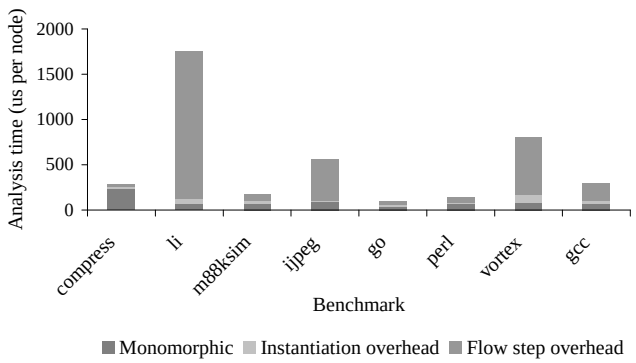


Figure 10: Running times

Test program	Code lines	AST nodes	Ave. deref size		Analysis time (secs)		
			Mono	Poly	Mono	Poly	Flow
compress	1,904	2,234	7	3	0.5	0.6	10.7%
li	7,602	23,379	282.1	185.2	1.8	40.9	93.2%
m88ksim	19,412	65,967	107.3	11.6	4.6	11.5	42.8%
ijpeg	31,215	79,486	37.9	11.1	7.0	44.8	80.7%
go	29,919	109,134	51.3	16.6	4.7	9.9	38.1%
perl	26,871	116,490	51.1	21.1	7.6	16.4	39.8%
vortex	67,211	200,107	1661.3	61.9	17.3	161.3	79.5%
gcc	205,406	604,100	429.5	91.8	42.3	179.5	64.7%

Table 1: Raw measurement data: Lines of code and AST node count. Average sizes of points-to sets at static dereference points, and running time in seconds.

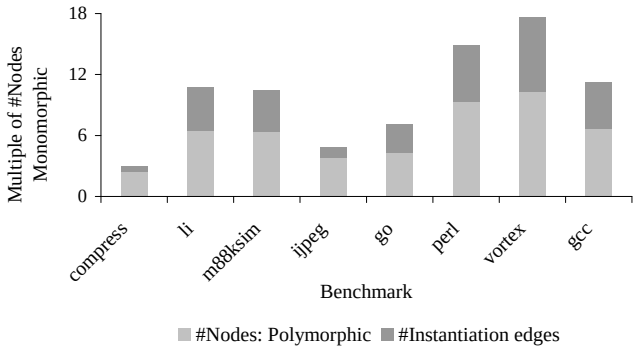


Figure 11: Space overhead

In contrast to the context-free language reachability for interprocedural precise flow paths of Reps al. [RHS95], valid paths in our analysis have the simple form of PosNeg-paths due to the absence of directional flow within a function. Along another dimension our flow is more general, since it deals directly with structured data types and higher-order programs.

Heintze and McAllester present a sub-transitive closure analysis for ML [HM97] also based on types. Like ours, their analysis traces flow paths on type graphs, but flow paths are not context-sensitive. Furthermore, their analysis requires a non-recursive type graph which precludes its application to C programs.

In Lackwit [OJ97], O'Callahan and Jackson exploit a relation induced by instantiations of polymorphic types, called *compatibility*. Compatibility is undirected and can be understood as a less precise version of our flow relation.

Foster et. al. [FFA00] presents a study of the relative precision trade-offs of monomorphic vs. polymorphic points-to analyses, both for directed and undirected intra-procedural flow. Their polymorphic version of Steensgaard's points-to analysis is less precise than the one presented here in two aspects: 1) sets of mutually recursive functions are analyzed monomorphically, and 2), the flow computation does not take full advantage of the polarities described here.

The analyses of Chatterjee et. al. [CRL99] and Wilson and Lam [WL95] are flow sensitive and much more precise than the analysis presented here. Their scalability remains unknown.

Finally, the technique presented here is a special case of a more general analysis based on both directional flow constraints and instantiation constraints [FRD00].

7 Conclusion

This paper argues that type-based context-sensitive analyses based on parametric polymorphism harbor implicit directional inter-procedural flow, even in the case where intra-procedural flow is undirected. The inter-procedural flow is defined by annotating instantiation edges with polarities and interpreting them as directed flow edges. We presented these ideas through a context-sensitive points-to analysis for C.

The resulting algorithm computes individual flow queries in linear time. We have presented empirical evidence supporting the practical nature of the approach.

References

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In OOPSLA'99 [OOP99], pages 1–19.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1999.

[DM82] L. Damas and R. Milner. Principle type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[FFA00] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. Polymorphic versus monomorphic points-to analysis. In *Proceedings of the 7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer Verlag, June 2000.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. From polymorphic subtyping to CFL reachability: Context-sensitive flow analysis using instantiation constraints. Technical Report MSR-TR-99-84, Microsoft Research, March 2000.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 32:6 in SIGPLAN notices, pages 261–272, June 1997.

[JW95] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the 2nd International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 207–224. Springer Verlag, September 1995.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[KTU94] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, March 1994.

[LH99] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1999.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MR97] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 74–89. ACM Press, June 1997.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Symposium on Programming*, pages 217–228, 1984.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 332–345. ACM Press, January 1997.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, May 1997.

[OOP99] *Proceedings of 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 34, 10 of *ACM SIGPLAN Notices*. ACM Press, November 1999.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 49–61, 1995.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 30:6 in SIGPLAN notices, June 1995.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In OOPSLA'99 [OOP99], pages 187–206.