Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C†

Jeffrey S. Foster[1], Manuel Fähndrich[2], and Alexander Aiken[1]

[1] University of California, Berkeley, 387 Soda Hall #1776, Berkeley, CA 94720
{jfoster,aiken}@cs.berkeley.edu
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052
maf@microsoft.com

Abstract We carry out an experimental analysis for two of the design dimensions of flow-insensitive points-to analysis for C: polymorphic versus monomorphic and equality-based versus inclusion-based. Holding other analysis parameters fixed, we measure the precision of the four design points on a suite of benchmarks of up to 90,000 abstract syntax tree nodes. Our experiments show that the benefit of polymorphism varies significantly with the underlying monomorphic analysis. For our equality-based analysis, adding polymorphism greatly increases precision, while for our inclusion-based analysis, adding polymorphism hardly makes any difference. We also gain some insight into the nature of polymorphism in points-to analysis of C. In particular, we find considerable polymorphism available in function parameters, but little or no polymorphism in function results, and we show how this observation explains our results.

1 Introduction

When constructing a constraint-based program analysis, the analysis designer must weigh the costs and benefits of many possible design points. Two important tradeoffs are:

- Is the analysis *polymorphic* or *monomorphic*? A polymorphic analysis separates analysis information by call site, while monomorphic analysis conflates all call sites. A polymorphic analysis is more precise but also more expensive than a corresponding monomorphic analysis.
- What is the underlying constraint relation? Possibilities include equalities (solved with unification) or more precise and expensive inclusions (solved with dynamic transitive closure), among many others.

Intuitively, if we want the greatest possible precision, we should use a polymorphic inclusion-based analysis, while if we are mostly concerned with efficiency, we should use a monomorphic equality-based analysis. But how much

† This research was supported in part by the National Science Foundation Young Investigator Award No. CCR-9457812, NASA Contract No. NAG2-1210, an NDSEG fellowship, and an equipment donation from Intel.

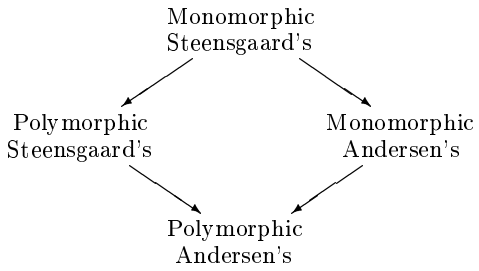


Figure 1. Relation between the four analyses. There is an edge from analysis x to analysis y if y is *at least as precise* as x.

more precision does polymorphism add, and what do we lose by using equality constraints? In this paper, we try to answer these questions for a particular constraint-based program analysis, *flow-insensitive points-to analysis for C*. Our goal is to compare the tradeoffs between the four possible combinations of polymorphism/monomorphism and equality constraints/inclusion constraints.

Points-to analysis computes, for each expression in a C program, a set of abstract memory locations (variables and heap) to which the expression could point. Our monomorphic inclusion-based analysis (Sect. 4.1) implements a version of Andersen's points-to analysis [4], and our monomorphic equality-based analysis (Sect. 4.2) implements a version of Steensgaard's points-to analysis [29]. To add polymorphism to Andersen's and Steensgaard's analyses (Sect. 4.3), we use Hindley-Milner style parametric polymorphism [21].

Our analyses are designed such that monomorphic Andersen's analysis is at least as precise as monomorphic Steensgaard's analysis [16, 28], and similarly for the polymorphic versions. Given the construction of our analyses, it is a theorem that the hierarchy of precision shown in Fig. 1 always holds. The main contribution of this work is the quantification of the exact relationship among these analyses. A secondary contribution of this paper is the development of polymorphic versions of Andersen's and Steensgaard's points-to analyses.

Running the analyses on our suite of benchmarks, we find the following results (see Sect. 5), where $\ll$ is read "is significantly less precise than." In general,

$$\begin{array}{c}\text{Monomorphic Steensgaard's} \ll \\ \text{Polymorphic Steensgaard's} \ll \\ \text{Polymorphic Andersen's}\end{array}$$

$$\begin{array}{c}\text{Monomorphic Steensgaard's} \ll \\ \text{Monomorphic Andersen's} \approx \\ \text{Polymorphic Andersen's}\end{array}$$

The exact relationships vary from benchmark to benchmark. These results are rather surprising—why should polymorphism not add much precision to Andersen's analysis but benefit Steensgaard's analysis? While we do not have definitive answers to these questions, Sect. 5.3 suggests some possible explanations.

Notice from this table that monomorphic Andersen's analysis is approximately as precise as polymorphic Andersen's analysis, while polymorphic Steensgaard's analysis is much less precise than polymorphic Andersen's analysis. Note, however, that polymorphic Steensgaard's analysis and monomorphic Andersen's analysis are in general incomparable (see Sect. 5.1). Still, given that polymorphic analyses are much more complicated to understand, reason about, and implement than their monomorphic counterparts, these results suggest that monomorphic Andersen's analysis may represent the best design choice among the four analyses. This may be a general principle: in order to improve a program analysis, developing a more powerful monomorphic analysis may be preferable to adding context-sensitivity, one example of which is Hindley-Milner style polymorphism.

Carrying out an experimental exploration of even a portion of the design space for non-trivial program analyses is a painstaking task. In interpreting our results there are two important things to keep in mind. First, our exploration of even the limited design space of flow-insensitive points-to analysis for C is still partial—there are dimensions other than the two that we explore that may not be orthogonal and may lead to different tradeoffs. For example, it may matter how precisely heap memory is modeled, how strings are modeled, whether C `structs` are analyzed by field or all fields are summarized together, and so on. Section 5 details our choices for these parameters. Also, Hindley-Milner style polymorphism is only one way to add context-sensitivity to a points-to analysis, and other approaches (e.g., polymorphic recursion [15]) may yield different tradeoffs.

Second, our experiments measure the relative precision of each analysis. They do not measure the relative impact of each analysis in a compiler. For example, it may be that some points-to sets are more important than others to an optimizer, and thus increases in precision may not always lead to better optimizations. However, a more precise analysis should not lead to worse optimizations than a less precise analysis. We should also point out that it is difficult to separate the benefit of a pointer analysis in a compiler from the design of the rest of the optimizer. Measures of relative precision have the advantage of being independent of the specific choices made in using the analysis information by a particular tool.

2 Related Work

Andersen's [4] and Steensgaard's [29] points-to analyses are only two choices in a vast array of possible alias analyses, among them [5, 6, 7, 8, 9, 10, 11, 15, 19, 20, 27, 28, 31, 33, 34]. As our results suggest, the benefit of polymorphism (more generally, *context-sensitivity*) may vary greatly with the particular analysis.

Hindley-Milner style polymorphism [21] has been studied extensively. The only direct applications of Hindley-Milner polymorphism to C of which we are

aware are the analyses in this paper, the polymorphic recursive analysis proposed in [15] (see below), and the Lackwit system [23]. Lackwit, a software engineering tool, computes ML-style types for C and appears to scale very well to large programs.

Mossin [22] develops a polymorphic flow analysis based on polymorphic recursion and atomic subtyping constraints. Mossin's system starts with a type-annotated program and infers atomic flow constraints, whereas we infer the type and flow annotations simultaneously and do not have an atomic subtyping system. [15] develops an efficient algorithm for both subtyping and equality-based polymorphic recursive flow analyses, and shows how to construct a polymorphic recursive version of Steensgaard's analysis. (In contrast, in this paper we use Hindley-Milner style polymorphism, which can be less precise.) We believe that the techniques of [15] can also be adapted to Andersen's analysis.

Other research has explored making monomorphic inclusion-based analyses scalable. [14] describes an online cycle-elimination algorithm for simplifying inclusion constraints. [30] describes a related optimization technique, *projection merging*, which merges multiple projections of the same set variable. Our current implementation uses both of these techniques, which makes it possible to run the polymorphic inclusion-based analysis on our larger benchmarks.

Finally, we discuss a selection of related analyses. Wilson and Lam [31] propose a flow-sensitive alias analysis that distinguishes calls to the same function in different aliasing contexts. Their system analyzes a function once for each aliasing pattern of its actual parameters. In contrast, we analyze each function only once, independently of its context, by constructing types that summarize functions' points-to effects in any context.

Ruf [26] studies the tradeoff between context-sensitivity and context-insensitivity for a particular dataflow-style alias analysis, discovering that context-sensitivity makes little appreciable difference in the accuracy of the results. Our results partially agree with his. For Andersen's inclusion-based analysis we find the same trend. However, for Steensgaard's equality-based analysis, which is substantially less precise than Ruf's analysis, adding polymorphism makes a significant difference

Emami, Ghiya, and Hendren [11] propose a flow-sensitive, context-sensitive analysis. The scalability of this analysis is unknown.

Landi and Ryder [20] study a very precise flow-sensitive, context-sensitive analysis. Their flow-sensitive system has difficulty scaling to large programs; recent work has focused on combined analyses that apply different alias analyses to different parts of a program [35].

Chatterjee, Ryder, and Landi [6] propose an analysis for Java and C++ that uses a flow-sensitive analysis with conditional points-to relations whose validity depends on the aliasing and type information provided by the context. While the style of polymorphism used in [6] appears related to Hindley-Milner style polymorphism, the exact relationship is unclear.

Das [7] proposes a monomorphic alias analysis with precision close to Andersen's analysis but cost close to Steensgaard's analysis. The effect of adding

polymorphism to Das's analysis is currently unknown but cannot yield more precision than polymorphic Andersen's analysis.

3 Constraints

Our analyses are formulated as non-standard type systems for C. We follow the usual approach for constraint-based program analysis: As the analyses infer types for a program's expressions, a system of typing constraints is generated on the side. The solution to the constraints defines the points-to graph of the program.

Our analyses are implemented with the Berkeley Analysis Engine (BANE) [1], which is a framework for constructing constraint-based analyses. BANE supports analyses involving multiple *sorts* of constraints, two of which are used by our points-to analyses. Our implementation of Andersen's analysis uses inclusion (or *set*) constraints [2, 18]. Our implementation of Steensgaard's analysis uses a mixture of equality (or *term*) and inclusion constraints. The rest of this section provides background on the constraint formalisms.

Each sort of constraint comes equipped with a constraint relation. The relation between set expressions is $\subseteq$, and the relation between term expressions is $=$. For our purposes, *set expressions* se consist of set variables $\mathcal{X}, \mathcal{Y}, \ldots$ from a family of variables *Vars* (caligraphic text denotes variables), terms constructed from n-ary constructors $c \in \mathit{Con}$, a special form $\mathit{proj}(c, i, se)$, an empty set 0, and a universal set 1.

$$se ::= \mathcal{X} \mid c(se_1, \ldots, se_n) \mid \mathit{proj}(c, i, se) \mid 0 \mid 1$$

Similarly, *term expressions* are of the form

$$te ::= \mathcal{X} \mid c(te_1, \ldots, te_n) \mid 0$$

Here 0 represents a special, distinguished nullary constructor.

Each constructor c is given a *signature* S_c specifying the arity, variance, and sort of c. If S is the set of sorts (in this case, $S = \{\mathbf{Term}, \mathbf{Set}\}$), then constructor signatures are of the form

$$c : \iota_1 \times \cdots \times \iota_{\mathrm{arity}(c)} \to S$$

where ι_i is s (covariant) or $\overline{s}$ (contravariant) for some $s \in S$. Intuitively, a constructor c is *covariant* in an argument $\mathcal{X}$ if the set denoted by a term $c(\ldots, \mathcal{X}, \ldots)$ becomes larger as $\mathcal{X}$ increases. Similarly, a constructor c is *contravariant* in an argument $\mathcal{X}$ if the set denoted by a term $c(\ldots, \mathcal{X}, \ldots)$ becomes smaller as $\mathcal{X}$ increases. To improve readability, we mark contravariant arguments with overbars.

One example constructor from Andersen's analysis is

$$\mathit{lam} : \mathbf{Set} \times \overline{\mathbf{Set}} \times \mathbf{Set} \to \mathbf{Set}$$

The *lam* constructor models function types. The first (covariant) argument names the function, the second (contravariant) argument represents the domain, and the third (covariant) argument represents the range.

Steensgaard's analysis uses a constructor

$$\mathit{ref} : \mathbf{Set} \times \mathbf{Term} \times \mathbf{Term} \to \mathbf{Term}$$

to model locations. The first field models the set of aliases of this location, and the second and third fields model the contents of this location. See Sect. 4.2 for a discussion of why a set is needed for the first field. More discussion of mixed constraints can be found in [12, 13].

Our system also includes *conditional equality constraints* $L \leq R$ (defined on terms) to support Steensgaard's analysis (see Sect. 4.2). The constraint $L \leq R$ holds if either $L = R$ or $L = 0$ holds. Intuitively, if L is ever unified with a constructed term, then the constraint $L \leq R$ becomes $L = R$. Otherwise $L \leq R$ makes no constraint on R.

Our language of set constraints has no explicit operation to select components of a constructor. Instead we use constraints of the form

$$L \subseteq c(\ldots, \mathcal{Y}_i, \ldots) \qquad (*)$$

to make $\mathcal{Y}_i$ contain $c^{-i}(L)$ if c is covariant in i, and to make $c^{-i}(L)$ contain $\mathcal{Y}_i$ if c is contravariant in i. However, such a constraint is inconsistent if L contains terms whose head constructor is not c. To overcome this limitation, we define constraints of the form

$$L \subseteq \mathit{proj}(c, i, \mathcal{Y}_i)$$

This constraint has the same effect as $(*)$ on the elements of L constructed with c, and no effect on the other elements of L.

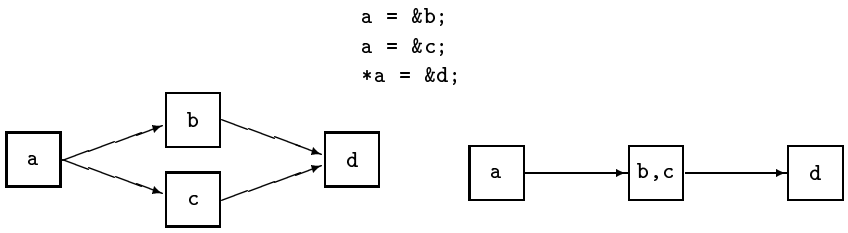
Solving a system of constraints involves computing an explicit *solved form* of all solutions or of a particular solution. See [3, 12, 13] for a thorough discussion of the constraint solver used in BANE.

4 The Analyses

This section develops monomorphic and polymorphic versions of Andersen's and Steensgaard's analyses. The presentation of the monomorphic version of Andersen's analysis mostly follows [14, 30] and is given primarily to make the paper self contained.

For a C program, points-to analysis computes a set of abstract memory locations (variables and heap) to which each expression could point. Andersen's and Steensgaard's analyses compute a *points-to graph* [11]. Graph nodes represent abstract memory locations, and there is an edge from a node x to a node y if x may contain a pointer to y. Informally, the analyses begin with some initial points-to relationships and close the graph under the rule

> For an assignment $e_1 = e_2$, anything in the points-to set for e_2 must also be in the points-to set for e_1.



(a) Andersen's Analysis (b) Steensgaard's Analysis

Figure 2. Example points-to graph

For Andersen's analysis, each node in the points-to graph may have directed edges to any number of other nodes. For Steensgaard's analysis, each node may have at most one out-edge, and graph nodes are coalesced if necessary to enforce this requirement. Figure 2 shows the points-to graph for a simple C program computed by Andersen's analysis (a) and Steensgaard's analysis (b).

4.1 Andersen's Analysis

In Andersen's analysis, types τ represent sets of abstract memory locations and are described by the following grammar:

$$\begin{aligned} \rho &::= \mathcal{P}_{\mathtt{x}} \mid l_{\mathtt{x}} \\ \tau &::= \mathcal{X} \mid \mathit{ref}(\rho, \tau, \overline{\tau}) \mid \mathit{lam}(\rho, \overline{\tau}, \tau) \end{aligned}$$

Here the constructor signatures are

$$\begin{aligned} \mathit{ref} &: \mathbf{Set} \times \mathbf{Set} \times \overline{\mathbf{Set}} \to \mathbf{Set} \\ \mathit{lam} &: \mathbf{Set} \times \overline{\mathbf{Set}} \times \mathbf{Set} \to \mathbf{Set} \end{aligned}$$

$\mathcal{X}$ and $\mathcal{P}_{\mathtt{x}}$ are set variables, and $l_{\mathtt{x}}$ is a constant (a constructor of arity 0). Contravariant arguments are marked with overbars. Note that function types $\mathit{lam}(\cdots)$ are contravariant in the domain (second argument) and covariant in the range (third argument).

Memory locations can be thought of as abstract data types with two operations, one to *get* the value stored in the location and one to *set* it. Intuitively, the *get* and *set* operations have types

- $\mathit{get} : \mathtt{void} \to \mathcal{X}$
- $\mathit{set} : \mathcal{X} \to \mathtt{void}$

where $\mathcal{X}$ is the type of data held in the memory location. Dereferencing a location corresponds to applying the *get* operation, and updating a location corresponds to applying the *set* operation. Note that the type variable $\mathcal{X}$ appears covariantly in the type of the *get* operation and contravariantly in the type of the *set* operation.

Translating this intuition into a set constraint formulation, the location of a variable x is modeled with the type $ref(l_{\mathtt{x}}, \mathcal{X}, \overline{\mathcal{X}})$, where $l_{\mathtt{x}}$ is a constant representing the name of the location, the covariant occurrence of $\mathcal{X}$ represents the *get* method, and the contravariant occurrence of $\mathcal{X}$ (marked with an overbar) represents the *set* method. For convenience, we choose not to represent the `void` components of the *get* and *set* methods' types.

We also associate with each location x a set variable $\mathcal{P}_{\mathtt{x}}$ and add the constraints $\mathcal{X} \subseteq proj(ref, 1, \mathcal{P}_{\mathtt{x}})$ and $\mathcal{X} \subseteq proj(lam, 1, \mathcal{P}_{\mathtt{x}})$. This constrains $\mathcal{P}_{\mathtt{x}}$ to contain the set of abstract locations, including functions, in the points-to set $\mathcal{X}$. The points-to graph is then defined by the least solution of $\mathcal{P}_{\mathtt{x}}$ for every location x. In the set formulation, the least solution for the points-to graph shown in Fig. 2a is

$$\mathcal{P}_{\mathtt{a}} = \{l_{\mathtt{b}}, l_{\mathtt{c}}\} \qquad \mathcal{P}_{\mathtt{b}} = \{l_{\mathtt{d}}\} \qquad \mathcal{P}_{\mathtt{c}} = \{l_{\mathtt{d}}\}$$

In addition to reference types we also must model function types, since C allows pointers to functions to be stored in memory. The type $lam(l_{\mathtt{f}}, \overline{\tau_1}, \tau_2)$ represents the function named f (every C function has a name) with argument τ_1 and return value τ_2. For simplicity the grammar allows only one argument. In our implementation, arguments are modeled with an ordered record $\{\tau_1, \ldots, \tau_n\}$ [25].[1]

Figure 3 shows a fragment of the type rules for the monomorphic version of Andersen's analysis. Judgments are of the form $A \vdash e : \tau; C$, meaning that in typing environment A, expression e has type τ under the constraints C. For simplicity we present only the interesting type rules. The full rules for all of C can be found in [16].

We briefly discuss the rules. To avoid having separate rules for l- and r-values, we model all variables as l-types. Thus the type of a variable x is our representation of its location, i.e., a *ref* type.

- Rule (Var_A) states that typings in the environment trivially hold.
- The address-of operator (Addr_A) adds a level of indirection to its operand by adding a *ref* constructor. The location (first) and *set* (third) fields of the resulting type are 0 and 1, respectively, because `&`e is not itself an l-value and cannot be updated.
- The dereferencing operator (Deref_A) removes a *ref* and makes the fresh variable $\mathcal{T}$ a superset of the points-to set of τ. Note the use of *proj* in case τ also contains a function type.
- The assignment rule (Asst_A) uses the same technique as (Deref_A) to *get* the contents of the right-hand side, and then uses the contravariant *set* field of the *ref* constructor to store the contents in the left-hand side location. See [16] for detailed explanations and examples.

[1] Note that we do not handle variable-length argument lists (varargs) correctly even with records. Handling varargs requires compiler- and architecture-specific knowledge of the layout of parameters in memory. See Sect. 5.

$$\frac{}{A \vdash \mathtt{x} : A(\mathtt{x}); \emptyset} \quad (\mathrm{Var}_A)$$

$$\frac{A \vdash e : \tau; C}{A \vdash \mathtt{\&}e : \mathit{ref}(0, \tau, \overline{1}); C} \quad (\mathrm{Addr}_A)$$

$$\frac{\begin{array}{c} A \vdash e : \tau; C \\ C' = C \ \wedge \ \tau \subseteq \mathit{proj}(\mathit{ref}, 2, \mathcal{T}) \end{array}}{A \vdash \mathtt{*}e : \mathcal{T}; C'} \quad (\mathrm{Deref}_A)$$

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1; C_1 \ A \vdash e_2 : \tau_2; C_2 \\ C = C_1 \ \wedge \ C_2 \ \wedge \\ \tau_1 \subseteq \mathit{proj}(\mathit{ref}, 3, \mathcal{T}) \ \wedge \\ \tau_2 \subseteq \mathit{proj}(\mathit{ref}, 2, \mathcal{T}) \end{array}}{A \vdash e_1 \mathtt{=} e_2 : \tau_2; C} \quad (\mathrm{Asst}_A)$$

$$\frac{A[\mathtt{x} \mapsto \mathit{ref}(l_{\mathtt{x}}, \mathcal{X}, \overline{\mathcal{X}})] \vdash e : \tau; C}{A \vdash \mathtt{let}\ x\ \mathtt{in}\ e\ \mathtt{ni} : \tau; C} \quad (\mathrm{LetRef}_A)$$

$$\frac{\begin{array}{c} \tau_f = \mathit{ref}(0, \mathit{lam}(l_{\mathtt{f}}, \overline{\mathcal{X}}, \mathcal{R}_{\mathtt{f}}), \overline{1}) \\ \tau_x = \mathit{ref}(l_{\mathtt{x}}, \mathcal{X}, \overline{\mathcal{X}}) \\ A[\mathtt{f} \mapsto \tau_f, \mathtt{x} \mapsto \tau_x] \vdash e : \tau; C \\ C' = C \ \wedge \ \tau \subseteq \mathit{proj}(\mathit{ref}, 2, \mathcal{R}_{\mathtt{f}}) \end{array}}{A \vdash \mathtt{fun\ f\ x\ =}\ e : \tau_f; C'} \quad (\mathrm{Lam}_A)$$

$$\frac{\begin{array}{c} A \vdash \mathtt{*}e_1 : \tau_1; C_1 \ A \vdash e_2 : \tau_2; C_2 \\ C = C_1 \ \wedge \ C_2 \ \wedge \\ \tau_2 \subseteq \mathit{proj}(\mathit{ref}, 2, \mathcal{T}) \ \wedge \\ \tau_1 \subseteq \mathit{proj}(\mathit{lam}, 2, \mathcal{T}) \ \wedge \\ \tau_1 \subseteq \mathit{proj}(\mathit{lam}, 3, \mathcal{R}) \end{array}}{A \vdash e_1\ e_2 : \mathit{ref}(0, \mathcal{R}, \overline{1}); C} \quad (\mathrm{App}_A)$$

Figure 3. Constraint generation rules for Andersen's analysis. $\mathcal{T}$ and $\mathcal{R}$ stand for fresh variables

- The rule (LetRef_A) introduces new variables. Since this is C, all variables are in fact updateable references, and we allow them to be uninitialized.
- The rule (Lam_A) defines a possibly-recursive function `f` whose result is e. We lift each function type to an l-type by adding a *ref* as in (Asst_A). For simplicity the C issues of promotions from function types to pointer types, and the corresponding issues with `*` and `&` applied to functions, are ignored. These issues are handled correctly by our implementation. Notice a function type contains the value of its parameter, $\mathcal{X}$, not a reference $\mathit{ref}(l_{\mathtt{x}}, \mathcal{X}, \overline{\mathcal{X}})$. Analogously the range of the function type is also a value.
- Function application (App_A) constrains the formal parameter of a function type to contain the actual parameter, and makes the return type of the function a lower bound on fresh variable $\mathcal{R}$. Notice the use of $\mathtt{*}e_1$ in the hypothesis of this rule, which we need because the function, an r-type, has

been lifted to an l-type in (Lam_S). The result $\mathcal{R}$, which is an r-type, is lifted to an l-type by adding a *ref* constructor, as in (Addr_A).

4.2 Steensgaard's Analysis

Intuitively, Steensgaard's analysis replaces the inclusion constraints of Andersen's analysis with equality constraints. The type language is a small modification of the previous system:

$$\begin{aligned}\rho &::= \mathcal{P}_{\mathtt{x}} \mid \mathcal{L}_{\mathtt{x}} \mid l_{\mathtt{x}} \\ \tau &::= \mathcal{X} \mid \mathit{ref}(\rho, \tau, \eta) \\ \eta &::= \mathcal{X} \mid \mathit{lam}(\tau, \tau)\end{aligned}$$

with constructor signatures

$$\begin{aligned}\mathit{ref} &: \mathbf{Set} \times \mathbf{Term} \times \mathbf{Term} \to \mathbf{Term} \\ \mathit{lam} &: \mathbf{Term} \times \mathbf{Term} \to \mathbf{Term}\end{aligned}$$

As before, ρ denotes locations and τ denotes updateable references. Following [29], in this system function types η are always structurally within $\mathit{ref}(\cdots)$ types because in a system of equality constraints we cannot express a union $\mathit{ref}(\ldots) \cup \mathit{lam}(\ldots)$. For a similar reason location sets ρ consist solely of variables $\mathcal{P}_{\mathtt{x}}$ or $\mathcal{L}_{\mathtt{x}}$ and are modeled as sets (see below).

Each program variable $\mathtt{x}$ is modeled with the type $\mathit{ref}(\mathcal{L}_{\mathtt{x}}, \mathcal{X}, \mathcal{F}_{\mathtt{x}})$, where $\mathcal{L}_{\mathtt{x}}$ is a **Set** variable. For each location $\mathtt{x}$ we add a constraint $l_{\mathtt{x}} \subseteq \mathcal{L}_{\mathtt{x}}$, where $l_{\mathtt{x}}$ is a nullary constructor (as in Andersen's analysis). We also associate with location $\mathtt{x}$ another set variable $\mathcal{P}_{\mathtt{x}}$ and add the constraint $\mathcal{X} \leq \mathit{ref}(\mathcal{P}_{\mathtt{x}}, *, *)$, where $*$ stands for a fresh unnamed variable.

We compute the points-to graph by finding the least solution of the $\mathcal{P}_{\mathtt{x}}$ variables. For the points-to graph in Fig. 2b, the result is

$$\mathcal{P}_{\mathtt{a}} = \{l_{\mathtt{b}}, l_{\mathtt{c}}\} \qquad \mathcal{P}_{\mathtt{b}} = \{l_{\mathtt{d}}\} \qquad \mathcal{P}_{\mathtt{c}} = \{l_{\mathtt{d}}\}$$

Notice that `b` and `c` are inferred to be aliased, i.e., $\mathcal{L}_{\mathtt{b}} = \mathcal{L}_{\mathtt{c}}$. If we had instead used nullary constructors directly in the ρ field of *ref*, or had the ρ field been a **Term** sort, then the constraints would have been inconsistent, since $l_{\mathtt{b}} \neq l_{\mathtt{c}}$.

In Steensgaard's formulation [29], the relation between locations $\mathtt{x}$ and their corresponding term variables $\mathcal{P}_{\mathtt{x}}$ is implicit. While this suffices for a monomorphic analysis, in a polymorphic analysis maintaining this map is problematic, as generalization, simplification, and instantiation (see Sect. 4.3) all cause variables to be renamed.

Mixed constraints provide an elegant solution to this problem. By explicitly representing the mapping from locations to location names in a constraint formulation, we guarantee that any sound constraint manipulations preserve this mapping.

Figure 4 shows the constraint generation rules for Steensgaard's analysis. The rules are similar to the rules for Andersen's analysis. Again, we briefly discuss the rules. As before, all variables are modeled as l-types.

$$\frac{}{A \vdash \mathtt{x} : A(\mathtt{x}); \emptyset} \quad (\mathrm{Var}_S)$$

$$\frac{A \vdash e : \tau; C}{A \vdash \mathtt{\&}e : \mathit{ref}(*, \tau, *); C} \quad (\mathrm{Addr}_S)$$

$$\frac{\begin{array}{c} A \vdash e : \tau; C \\ C' = C \ \wedge \ \tau \leq \mathit{ref}(*, \mathcal{T}, *) \end{array}}{A \vdash \mathtt{*}e : \mathcal{T}; C'} \quad (\mathrm{Deref}_S)$$

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1; C_1 \qquad A \vdash e_2 : \tau_2; C_2 \\ C = C_1 \ \wedge \ C_2 \ \wedge \\ \tau_1 \leq \mathit{ref}(*, \mathcal{T}_1, *) \ \wedge \ \tau_2 \leq \mathit{ref}(*, \mathcal{T}_2, *) \ \wedge \\ \mathcal{T}_2 \leq \mathcal{T}_1 \end{array}}{A \vdash e_1 \mathtt{=} e_2 : \tau_2; C} \quad (\mathrm{Asst}_S)$$

$$\frac{A[\mathtt{x} \mapsto \mathit{ref}(\mathcal{L}_\mathtt{x}, \mathcal{X}, \mathcal{F}_\mathtt{x})] \vdash e : \tau; C}{A \vdash \mathtt{let}\ x\ \mathtt{in}\ e\ \mathtt{ni} : \tau; C} \quad (\mathrm{LetRef}_S)$$

$$\frac{\begin{array}{c} \tau_f = \mathit{ref}(*, \mathit{ref}(\mathcal{L}_\mathtt{f}, \mathcal{T}_\mathtt{f}, \mathit{lam}(\mathcal{X}, \mathcal{R}_\mathtt{f})), *) \\ \tau_x = \mathit{ref}(\mathcal{L}_\mathtt{x}, \mathcal{X}, \mathcal{F}_\mathtt{x}) \\ A[\mathtt{f} \mapsto \tau_f, \mathtt{x} \mapsto \tau_x] \vdash e : \tau; C \\ C' = C \ \wedge \ \tau \leq \mathit{ref}(*, \mathcal{T}, *) \ \wedge \ \mathcal{T} \leq \mathcal{R}_\mathtt{f} \end{array}}{A \vdash \mathtt{fun\ f\ x\ =}\ e : \tau_f; C'} \quad (\mathrm{Lam}_S)$$

$$\frac{\begin{array}{c} A \vdash \mathtt{*}e_1 : \tau_1; C_1 \qquad A \vdash e_2 : \tau_2; C_2 \\ C = C_1 \ \wedge \ C_2 \ \wedge \\ \tau_1 \leq \mathit{ref}(*, *, \mathcal{F}) \ \wedge \ \mathcal{F} \leq \mathit{lam}(\mathcal{Y}, \mathcal{R}) \ \wedge \\ \tau_2 \leq \mathit{ref}(*, \mathcal{T}, *) \ \wedge \ \mathcal{T} \leq \mathcal{Y} \end{array}}{A \vdash e_1\ e_2 : \mathit{ref}(*, \mathcal{R}, *); C} \quad (\mathrm{App}_S)$$

Figure 4. Constraint generation rules for Steensgaard's analysis. $\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2, \mathcal{Y}$, and $\mathcal{R}$ are fresh variables. Each occurrence of $*$ is a fresh, unnamed variable

- Rules (Var_S) and (LetRef_S) are unchanged from Andersen's analysis.
- Rule (Addr_S) adds a level of indirection to its operand.
- Rule (Deref_S) removes a *ref* and makes fresh variable $\mathcal{T}$ contain the points-to set of τ.
- The assignment rule (Asst_S) makes fresh variables $\mathcal{T}_i$ contain the points-to sets of each e_i. (Asst_S) conditionally equates $\mathcal{T}_1$ with $\mathcal{T}_2$, i.e., if e_2 is a pointer, its points-to set is unified with the points-to set of e_1. Using conditional unification increases the accuracy of the analysis [29].
- Function definition (Lam_S) behaves as in Andersen's analysis. Here, $\mathit{ref}(\mathcal{L}_\mathtt{f}, \mathcal{T}_\mathtt{f}, \mathit{lam}(\mathcal{X}, \mathcal{R}_\mathtt{f}))$ represents the function type and the outermost *ref* lifts the function type to an l-type. Again a function type contains the r-types of its parameter and return value rather than their l-types. Notice that the type of the function `f` points to is stored in the second (τ) field of `f`'s type $\tau_\mathtt{f}$, not in the third (η) field. Thus in the assignment rule (Asst_S), the $\mathcal{T}_i$ variables contain both the functions and memory locations that the e_i point to.

$$\frac{A \vdash e : \tau; C \quad \vec{\mathcal{X}} \notin \mathrm{fv}(A)}{A \vdash e : \forall \vec{\mathcal{X}}.\tau \backslash C; C} \qquad \text{(Quant)}$$

$$\frac{A \vdash e : \forall \vec{\mathcal{X}}.\tau \backslash C'; C \quad \vec{\mathcal{Y}} \text{ fresh}}{A \vdash e : \tau[\vec{\mathcal{X}} \mapsto \vec{\mathcal{Y}}]; C \wedge C'[\vec{\mathcal{X}} \mapsto \vec{\mathcal{Y}}]} \qquad \text{(Inst)}$$

Figure 5. Rules for quantification

– Function application (App_S) conditionally equates the formal and actual parameters of a function type and evaluates to the return type. Note the use of $*e_1$ in the hypothesis of this rule, which is needed since the function type has been lifted to an l-type. Intuitively, this rule expands the application (`fun f x` $= e$) e_2 into the sequence `x` $= e_2$; e.

4.3 Adding Polymorphism

This section describes how the monomorphic analyses are extended to polymorphic analyses. While ultimately we find polymorphism unprofitable for our points-to analyses, this section documents a number of practical insights for the implementation of polymorphism in analysis systems considerably more elaborate than the Hindley/Milner system.

The rules in Figs. 3 and 4 track the constraints generated in the analysis of each expression. The monomorphic analyses have one global constraint system. In the polymorphic analyses, each function body has a distinct constraint system.

We introduce polymorphic constrained types of the form $\forall \vec{\mathcal{X}}.\tau \backslash C$. The type $\forall \vec{\mathcal{X}}.\tau \backslash C$ represents any type of the form $\tau[\vec{\mathcal{X}} \mapsto \vec{se}]$ under constraints $C[\vec{\mathcal{X}} \mapsto \vec{se}]$, for any choice of $\vec{se}$. Figure 5 shows the additional rules for quantification. The notation $\mathrm{fv}(A)$ stands for the free variables of environment A. Rule (Quant) states that we may quantify a type over any variables not free in the type environment. (Inst) allows us to instantiate a quantified type with fresh variables, adding the constraints from the quantified type to the system. These rules are standard [24].

We restrict quantification to non-*ref* types to avoid well-known problems with mixing updateable references and polymorphism [32]. In practical terms, this means that after analyzing a function definition, we can quantify over its parameters and its return value. The rule (Inst) says that we may instantiate a quantified type with fresh variables, adding the constraints from the quantified type to the environment.

If used naïvely, rule (Quant) amounts to analyzing a program in which all function calls have been inlined. In order to make the polymorphic analyses tractable, we perform a number of simplifications to reduce the sizes of quantified types. See [17] for a discussion of the simplifications we use.

As an example of the potential benefit of polymorphic points-to analysis, consider the following atypical C program:

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

```
int main() {
    int a, b, *c, *d;
    c = id(&a); d = id(&b);
}
```

In the notation in this paper `id` is defined as `fun id x = x`. In monomorphic Andersen's analysis all inputs to `id` flow to all outputs. Thus we discover that `c` and `d` both point to `a` and `b`. Polymorphic Andersen's analysis assigns `id` type

$$\forall \mathcal{X}, \mathcal{R}_{\mathtt{id}}.\, lam(l_{\mathtt{id}}, \overline{\mathcal{X}}, \mathcal{R}_{\mathtt{id}}) \backslash \\ ref(l_{\mathtt{x}}, \mathcal{X}, \overline{\mathcal{X}}) \subseteq proj(ref, 2, \mathcal{R}_{\mathtt{id}})$$

Solving these constraints and simplifying (see [17]) yields

$$\forall \mathcal{X}.\, lam(l_{\mathtt{id}}, \overline{\mathcal{X}}, \mathcal{X}) \backslash \emptyset$$

In other words, `id` is the identity function. Because this type is instantiated for each call of `id`, the points-to sets are computed exactly: `c` points to `a` and `d` points to `b`.

There are several important observations about the type system. First, function pointers do not have polymorphic types. Consider the following example:

```
int *f(...) { ... }
int foo(int *(*g)()) { x = g(...); y = g(...); z = f(...); }
int main() { foo(f); }
```

Within the body of `foo`, the type of `g` appears in the environment (with a monomorphic type), so variables in the type of `g` cannot be quantified. Hence both calls to `g` use the same instance of `f`'s type. The call directly through `f` can use a polymorphic type for `f`, and hence is to a fresh instance.

Second, we do not allow the types of mutually recursive functions to be polymorphic within the recursive definition. Thus we analyze sets of mutually recursive functions monomorphically and then generalize the types afterwards.

Finally, we require that function definitions be analyzed before function uses. We formally state this requirement using the following definition:

Definition 1. The *function dependence graph (FDG)* of a program is a graph $G = (V, E)$ with vertices V and edges E. V is the set of all functions in the program, and there is an edge in E from `f` to `g` iff function `f` contains an occurrence of the name of `g`.

A function's successors in the FDG for a program must be analyzed before the function itself. Note that the FDG is trivial to compute from the program text.

Figure 6 shows the algorithm for analyzing a program polymorphically. Each strongly-connected component of the FDG is visited in final depth-first order. We analyze each mutually-recursive component monomorphically and then apply quantification. We merge the simplified system C' into the top-level constraint system *Glob*, replacing *Glob* by $Glob \wedge C'$. Notice that we do not require a call graph for the analysis, but only the FDG, which is statically computable.

1. Make a fresh global constraint system *Glob*
2. Construct the function dependence graph G
3. For each non-root strongly-connected component S of G in final depth-first order
3a. Make a fresh constraint system C
3b. Analyze each $\texttt{f} \in S$ monomorphically in C
3c. Quantify each $\texttt{f} \in S$ in C, applying simplifications
3d. Compute $C' = C$ simplified and merge C' into *Glob*
4. Analyze the root SCC in *Glob*

Figure 6. Algorithm 1: Bottom-up pass

4.4 Reconstructing Local Information

After applying the bottom-up pass of Fig. 6, the analysis has correctly computed the points-to graph for the global variables and the local variables of the outermost function, usually called `main`. (There is no need to quantify the type of `main`, since its type can only be used monomorphically.) At this point we have lost alias information for local variables, for two reasons. First, applying simplifications during the analysis may eliminate the points-to variables corresponding to local variables completely. Second, whenever we apply (Inst) to instantiate the type of a function `f`, we deliberately lose information about the types of `f`'s local variables by replacing their points-to type variables with fresh type variables.

The points-to set of a local variable depends on the context(s) in which `f` is used. To reconstruct points-to information for locals, we keep track of the instantiated types of functions and use these to flow context information back into the original, unsimplified constraint system.

Figure 7 gives the algorithm for reconstructing the points-to information for the local variables of function `f` on a particular path or set of paths P in the FDG. Note that Algorithm 2 requires $\texttt{f} \in P$. The constraints given are for Andersen's analysis. For Steensgaard's analysis we replace $\subseteq$ constraints by the appropriate $\leq$ constraints. (Note that for Steensgaard's analysis there may be more precise ways of computing summary information. See [15].) In Algorithm 2, the constraint systems along the FDG path are merged into a fresh constraint system, and then the types of the actual parameters from each instance are linked to the types of the formal parameters of the original type. We also link the return values of the original type to the return values of the instances.

This algorithm computes the points-to sets for the local variables of `f` along FDG path P. Because this algorithm is parameterized by the FDG path, it lets the analysis client choose the precision of the desired information. An interactive software engineering tool may be interested in a particular use of a function (corresponding to a single path from `f` to the root), while a compiler, which must produce code that works for all instances, would most likely be interested in all paths from `f` to the root of the FDG.

In our experiments (Sect. 5), to compute information for function `f` we choose P to be all of `f`'s ancestors in the FDG. This corresponds exactly to a points-to

1. Let $C = Glob \wedge \bigwedge_{\mathtt{g} \in P} C_{\mathtt{g}}$ be a fresh system
2. For each function $\mathtt{g} \in P$
2a. Let $lam(l_{\mathtt{g}}, \overline{\mathcal{G}_1}, \mathcal{R}_1), \ldots, lam(l_{\mathtt{g}}, \overline{\mathcal{G}_n}, \mathcal{R}_n)$ be the instances of g's function type.
2b. Let $lam(l_{\mathtt{g}}, \overline{\mathcal{G}}, \mathcal{R})$ be g's original function type
2c. Add constraints $\mathcal{G}_i \subseteq \mathcal{G}$ and $\mathcal{R} \subseteq \mathcal{R}_i$ for $i \in [1..n]$.
3. Compute the points-to sets for f's locals in C.

Figure 7. Algorithm 2: Top-down pass for function f on FDG path or set of FDG paths P

analysis in which f and its ancestors are monomorphic and all other functions are polymorphic. Clearly there are cases in which this choice will lead to a loss of precision. However, the other natural alternative, to compute alias information for each of f's instances separately, would yield an exponential algorithm. By treating f monomorphically, in an FDG of size n Algorithm 2 requires copying $O(n^2)$ (unsimplified) constraint systems.

5 Experiments

We have implemented our analyses using BANE [1]. BANE manages the details of constraint representation and solving, quantification, instantiation, and simplification. Our analysis tool generates constraints and decides when and what to quantify, instantiate, and simplify.

Our analysis handles almost all features of C, following [29]. The only exceptions are that we do not correctly model expressions that rely on compiler-specific choices about the layout of data in memory, e.g., variable-length argument lists or absolute addressing.

Our experiments cover the four possible combinations of polymorphism (polymorphic or monomorphic) and analysis precision (inclusion-based or equality-based). Table 1 lists the suite of C programs on which we performed the analyses.[2] The size of each program is listed in terms of preprocessed source lines and number of AST nodes. The AST node count is restricted to those nodes the analysis traverses, e.g., this count ignores declarations.

As with most C programs, our benchmark suite makes extensive use of standard libraries. After analyzing each program we also analyze a special file of hand-coded stubs modeling the points-to effects of all library functions used by our benchmark suite. These stubs are not included in the measurements of points-to set sizes, and we only process the stubs corresponding to library functions that are actually used by the program. The stubs are modeled in the same way that regular functions are modeled. Thus they are treated monomorphically in the monomorphic analyses, and polymorphically in the polymorphic analyses.

[2] We modified the `tar-1.11.2` benchmark to use the built-in malloc rather than a user-defined malloc in order to model heap usage more accurately.

Table 1. Benchmark programs

Name	AST Nodes	Preproc Lines	Name	AST Nodes	Preproc Lines
allroots	700	426	less-177	15179	11988
diff.diffh	935	293	li	16828	5761
anagram	1078	344	flex-2.4.7	29960	9345
genetic	1412	323	pmake	31148	18138
ks	2284	574	make-3.72.1	36892	15213
ul	2395	441	tar-1.11.2	38795	17592
ft	3027	1180	inform-5.5	38874	12957
compress	3333	651	sgmls-1.1	44533	30941
ratfor	5269	1532	screen-3.5.2	49292	23919
compiler	5326	1888	cvs-1.3	51223	31130
assembler	6516	2980	espresso	56938	21537
ML-typecheck	6752	2410	gawk-3.0.3	71140	28326
eqntott	8117	2266	povray-2.2	87391	59689
simulator	10946	4216			

To model heap locations, we generate a fresh global variable for each syntactic occurrence of a `malloc`-like function in a program. In certain cases it may be beneficial to distinguish heap locations by call path, though we did not perform this experiment. We model structures as atomic, i.e., every field of a structure shares the same location. Recent results [33] suggest some efficient alternative approaches.

For the polymorphic analyses, when we apply Algorithm 2 (Fig. 7) to compute the analysis results for function `f`, we choose P to be the set of all paths from `f` to the root of the FDG.

5.1 Precision

Figures 8 and 9 graph for each benchmark the average size of the points-to sets at the dereference sites in the program. A higher average size indicates lower precision. Missing data points indicate that the analysis exceeded the memory capacity of the machine (2GB).

We also measure the precision of the analyses both when each string is modeled as a distinct location and when strings are completely ignored (modeled as 0). Note the different scales on different graphs. For the purposes of this experiment, functions are not counted in points-to sets, and multi-level dereferences are counted separately (e.g., in `**x` there are two dereferences). Array indexing on known arrays (expressions of type array) is not counted as dereferencing.

Table 2 gives the numeric values graphed in Figs. 8 and 9 and more detailed information about the distribution of points-to sets. Due to lack of space, we only give the data for the experiments that model strings as distinct locations. See [17] for the data when strings are modeled as 0. For each analysis style, we list the running time, the average points-to set sizes at dereference sites, and

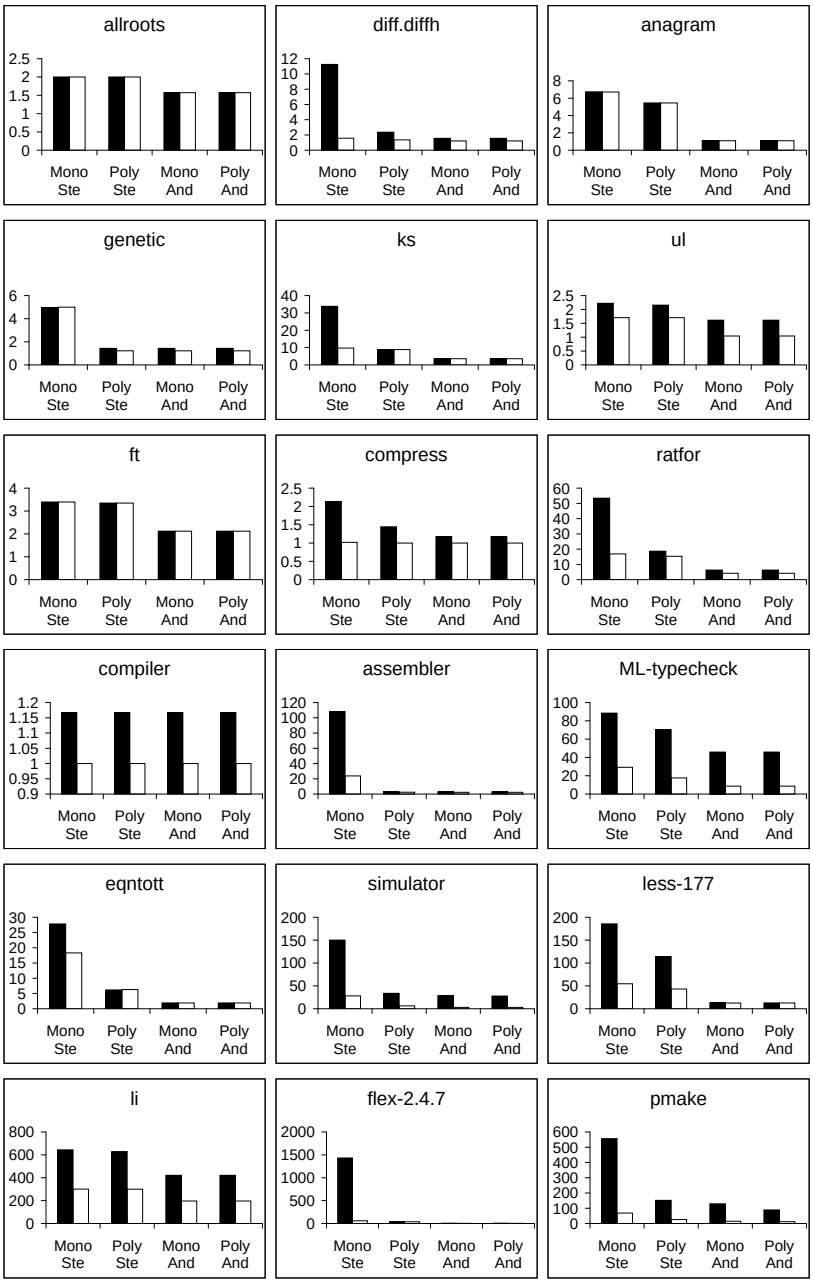


Figure 8. Average points-to sizes at dereference sites. The black bars give the results when strings are modeled; the white bars give the results when strings are not modeled

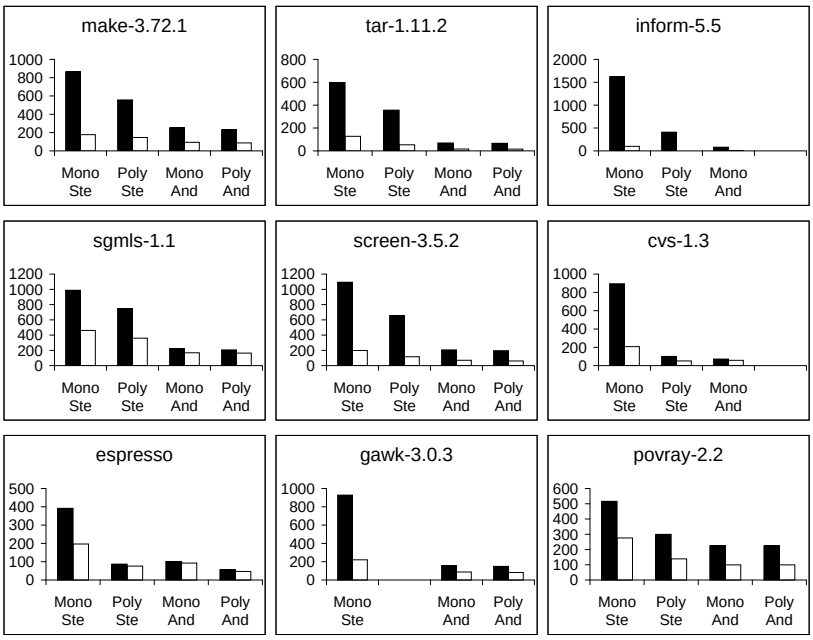


Figure 9. Continuation of Fig. 8. Average points-to sizes at dereference sites. The black bars give the results when strings are modeled; the white bars give the results when strings are not modeled

the number of dereference sites with points-to sets of size 1, 2, and 3 or more, plus the total number of non-empty dereference sites. (Most programs have some empty dereference sites because of dead code.) We also list the size of the largest points-to set.

Recall from the introduction that for a given dereference site, it is a theorem that the points-to sets computed by the four analyses are in the inclusion relations shown in Fig. 1. More precisely, there is an edge from analysis x in Fig. 1 to analysis y if for each expression e, the points-to set computed for e by analysis x contains the points-to set computed for e by analysis y. Two issues arise when interpreting the average points-to set size metric. First, when two analyses are related by inclusion the average size of points-to sets is a valid measure of precision. Thus we can use our metric to compare any two analyses *except* polymorphic Steensgaard's analysis and monomorphic Andersen's analysis.

For these two analyses there is no direct inclusion relationship. For a given expression e, if e_S is the points-to set computed by polymorphic Steensgaard's analysis and e_A is the points-to set computed by monomorphic Andersen's analysis, it may be that $e_S \not\subseteq e_A$ and $e_S \not\supseteq e_A$. Detailed examination of the points-to sets computed by polymorphic Steensgaard's analysis and monomorphic Andersen's analysis reveals that this does occur in practice, and thus the two analyses

Table 2. Data for string modeling experiments graphed in Fig. 8. The running times are the average of three for the monomorphic experiments, while the polymorphic experiments were only performed once.

Name	Monomorphic Steensgaard's							Polymorphic Steensgaard's							
	Time (s)	Av.	Num. deref sites					Up Tm (s)	Dn Tm (s)	Av.	Num. deref sites				
			1	2	3+	tot	max				1	2	3+	tot	max
allroots	0.17	2.00	0	42	0	42	2	0.27	0.29	2.00	0	42	0	42	2
diff.diffh	0.23	11.25	12	1	23	36	17	0.29	0.55	2.36	14	13	9	36	5
anagram	0.25	6.74	11	1	30	42	9	0.37	1.00	5.45	12	0	30	42	8
genetic	0.36	4.95	22	8	46	76	15	0.45	1.18	1.43	62	10	4	76	10
ks	0.43	33.83	3	13	99	115	39	0.53	1.38	8.86	3	13	99	115	10
ul	0.49	2.22	55	129	54	238	4	0.59	2.97	2.16	55	137	46	238	4
ft	0.65	3.39	29	8	133	170	4	1.05	4.58	3.35	37	0	133	170	4
compress	0.73	2.13	181	44	36	261	8	0.94	5.32	1.44	181	44	36	261	3
ratfor	1.65	53.41	36	4	125	165	80	2.71	30.90	18.65	36	7	122	165	62
compiler	1.15	1.17	65	13	0	78	2	2.47	5.76	1.17	65	13	0	78	2
assembler	2.54	108.03	79	31	273	383	213	5.22	58.96	2.98	223	36	124	383	120
ML-typecheck	2.92	88.41	28	0	285	313	97	3.92	60.87	70.33	28	27	258	313	85
eqntott	2.70	27.82	68	110	436	614	42	3.45	54.17	6.17	76	133	405	614	11
simulator	3.78	150.11	24	13	259	296	223	5.70	118.20	33.71	105	5	186	296	89
less-177	5.66	185.55	69	13	490	572	219	18.28	321.89	114.13	80	14	478	572	173
li	18.67	643.88	8	0	933	941	657	33.33	695.71	629.01	8	0	933	941	644
flex-2.4.7	64.33	1431.68	13	0	1613	1626	1445	22.09	818.25	43.83	15	2	1609	1626	1226
pmake	20.98	556.19	40	2	2501	2543	570	373.97	4416.16	151.69	100	9	2434	2543	218
make-3.72.1	40.05	863.25	90	222	3170	3482	975	265.43	1045.70	556.94	311	158	3013	3482	666
tar-1.11.2	26.10	597.13	87	70	2031	2188	656	23.16	776.65	356.20	183	114	1888	2185	434
inform-5.5	47.81	1618.62	21	0	1268	1289	1648	2601.61	67608.52	408.47	28	0	1261	1289	601
sgmls-1.1	69.70	987.71	96	11	2382	2489	1046	126.08	3961.22	749.20	123	15	2351	2489	867
screen-3.5.2	64.79	1093.00	27	9	4915	4951	1110	65.37	1991.28	656.86	112	36	4803	4951	768
cvs-1.3	47.42	894.44	97	680	2276	3053	1242	124.80	2949.33	100.18	1159	141	1753	3053	367
espresso	34.40	391.59	101	530	5479	6110	456	104.65	3368.75	86.78	1238	595	4277	6110	171
gawk-3.0.3	78.30	927.57	139	50	4930	5119	966	—	—	—	—	—	—	—	—
povray-2.2	64.72	515.85	761	407	8044	9212	618	111.38	6606.45	299.41	1027	659	7526	9212	434

Name	Monomorphic Andersen's							Polymorphic Andersen's							
	Time (s)	Av.	Num. deref sites					Up Tm (s)	Dn Tm (s)	Av.	Num. deref sites				
			1	2	3+	tot	max				1	2	3+	tot	max
allroots	0.18	1.57	18	24	0	42	2	0.14	0.22	1.57	18	24	0	42	2
diff.diffh	0.18	1.56	25	2	9	36	3	0.21	0.49	1.56	25	2	9	36	3
anagram	0.24	1.10	38	4	0	42	2	0.16	0.72	1.10	38	4	0	42	2
genetic	0.22	1.43	62	10	4	76	10	0.21	0.76	1.43	62	10	4	76	10
ks	0.37	3.58	9	22	84	115	5	0.33	0.98	3.58	9	22	84	115	5
ul	0.24	1.61	184	8	46	238	4	0.23	0.91	1.61	184	8	46	238	4
ft	0.42	2.12	75	0	95	170	3	0.56	2.25	2.12	75	0	95	170	3
compress	0.34	1.18	215	46	0	261	2	0.41	1.42	1.18	215	46	0	261	2
ratfor	0.63	6.27	56	9	100	165	47	1.22	5.99	6.27	56	9	100	165	47
compiler	0.57	1.17	65	13	0	78	2	0.96	5.07	1.17	65	13	0	78	2
assembler	1.07	2.87	225	36	122	383	120	3.02	80.46	2.87	225	36	122	383	120
ML-typecheck	0.99	45.87	101	30	182	313	78	1.79	14.81	45.87	101	30	182	313	78
eqntott	1.03	1.92	239	199	176	614	5	1.50	11.20	1.92	239	199	176	614	5
simulator	1.35	28.53	107	10	179	296	72	2.32	51.70	27.78	107	10	179	296	71
less-177	2.55	12.98	221	92	259	572	110	4.35	184.03	12.72	238	101	233	572	110
li	4.44	421.23	28	0	913	941	465	189.49	9929.88	421.23	28	0	913	941	465
flex-2.4.7	4.81	6.22	734	204	688	1626	1226	8.61	173.97	6.21	735	204	687	1626	1226
pmake	5.11	129.16	401	98	2044	2543	175	21.38	682.71	88.64	452	98	1993	2543	144
make-3.72.1	9.02	250.85	619	268	2595	3482	494	13.18	390.35	230.12	652	264	2566	3482	487
tar-1.11.2	6.89	69.07	330	741	1117	2188	200	7.74	327.48	66.11	336	742	1107	2185	194
inform-5.5	6.95	80.51	657	20	612	1289	227	—	—	—	—	—	—	—	—
sgmls-1.1	8.14	224.11	687	321	1481	2489	506	40.52	1121.89	205.63	703	323	1463	2489	492
screen-3.5.2	7.45	206.48	339	39	4573	4951	241	1277.15	2028.85	195.83	342	44	4565	4951	232
cvs-1.3	10.82	71.27	1281	192	1580	3053	203	—	—	—	—	—	—	—	—
espresso	12.89	101.21	1824	300	3986	6110	175	28.81	967.64	56.34	1973	304	3833	6110	152
gawk-3.0.3	12.40	157.28	1177	226	3716	5119	237	22.14	763.62	148.77	1184	228	3707	5119	225
povray-2.2	22.40	223.61	2474	588	6150	9212	402	169.51	5574.82	223.61	2474	588	6150	9212	402

are incomparable in our metric. The best we can do is observe that monomorphic Andersen's analysis is almost as precise as polymorphic Andersen's analysis, and polymorphic Steensgaard's analysis is less precise than polymorphic Andersen's analysis.

Second, it is possible for a polymorphic analysis to determine that a monomorphically non-empty points-to set is in fact empty, and thus have a larger average points-to set size than its monomorphic counterpart (since only non-empty

points-to sets are included in this average). However, we can eliminate this possibility by counting the total number of nonempty dereference sites. (A polymorphic analysis cannot have more nonempty dereference sites than its monomorphic counterpart.) The data in Table 2 shows that for all benchmarks except `tar-1.11.2`, the total number of non-empty dereference sites is the same across all analyses, and the difference between the polymorphic and monomorphic analyses for `tar-1.11.2` is miniscule. Therefore we know that averaging the sizes of non-empty dereference sites is a valid measure of precision.

5.2 Speed

Table 2 also lists the running times for the analyses. The running times include the time to compute the least model of the $\mathcal{P}_{\mathtt{x}}$ variables, i.e., to find the points-to sets. For the polymorphic analyses, we separate the running times into the time for the bottom-up pass and the time for the top-down pass.

For purposes of this experiment, whose goal is to compare the precision of monomorphic and polymorphic points-to analysis, the running times are largely irrelevant. Thus we have made little effort to make the analyses efficient, and the running times should all be taken with a grain of salt.

5.3 Discussion

The data presented in Figs. 8 and 9 and Table 2 shows two striking and consistent results:

1. Polymorphic Andersen's analysis is hardly more precise than monomorphic Andersen's analysis.
2. Polymorphic Steensgaard's analysis is much more precise than monomorphic Steensgaard's analysis.

The only exceptions to these trends are some of the smaller programs (`allroots`, `ul`, `ft`, `compiler`, `li`), for which polymorphic Steensgaard's analysis is not much more precise than monomorphic Steensgaard's analysis, and one larger program, `espresso`, for which Polymorphic Andersen's analysis is noticeably more precise than Monomorphic Andersen's analysis. Additionally, notice that for all programs except `espresso`, polymorphic Steensgaard's analysis has a higher average points-to set size than monomorphic Andersen's analysis. (Recall that this does not necessarily imply strictly increased precision.)

To understand these results, consider the following code skeleton:

```
void f() { ... h(a); ... }
void g() { ... h(b); ... }
void h(int *c) { ... }
```

In Steensgaard's equality-based monomorphic analysis, the types of all arguments for all calls sites of a function are equated. In the example, this results in $a = b = c$, where a is `a`'s points-to type, b is `b`'s points-to type, and c is `c`'s

Table 3. Potential polymorphism. The measurements include library functions.

Name	Call Sites	% Void	Name	Call Sites	% Void
allroots	55	69	less-177	1091	56
diff.diffh	67	58	li	1243	37
anagram	59	75	flex-2.4.7	1205	79
genetic	79	75	pmake	1943	56
ks	101	84	make-3.72.1	1955	50
ul	103	74	tar-1.11.2	1586	54
ft	152	70	inform-5.5	2593	72
compress	138	73	sgmls-1.1	1614	62
ratfor	306	75	screen-3.5.2	2632	75
compiler	448	89	cvs-1.3	3036	55
assembler	519	66	espresso	2729	51
ML-typecheck	430	31	gawk-3.0.3	2358	51
eqntott	364	61	povray-2.2	3123	59
simulator	677	75			

points-to type. In the polymorphic version of Steensgaard's analysis, a and b can be distinct. Our measurements show that separating function parameters is important for points-to analysis.

In contrast, in Andersen's monomorphic inclusion-based system, the points-to types of arguments at call sites are potentially separated. In the example, we have $a \subseteq c$ and $b \subseteq c$. However, function results are all conflated (i.e., every call site has the same result, the union of points-to results over all call sites). The fact that polymorphic Andersen's analysis is hardly more precise than monomorphic Andersen's analysis suggests that separating function parameters is by far the most important form of polymorphism present in points-to analysis for C.

Thus, we conclude that polymorphism for points-to analysis is useful primarily for separating inputs, which can be achieved very nearly as well by a monomorphic inclusion-based analysis. This conclusion begs the question: Why is there so little polymorphism in points-to results available in C? Directly measuring the polymorphism available in output side effects of C functions is difficult, although we hypothesize that C functions tend to side-effect global variables and heap data (which our analyses model as global) rather than stack-allocated data.

We can measure the polymorphism of result types fairly directly. Table 3 lists for each benchmark the number of call sites and percentage of calls that occur in void contexts. These results emphasize that most C functions are called for their side effects: for 25 out of 27 benchmarks, at least half of all calls are in void contexts. Thus, there is a greatly reduced chance that polymorphism can be beneficial for Andersen's analysis.

It is worth pointing out that the client for a points-to analysis can also have a significant, and often negative, impact on the polymorphism that actually can be exploited. In the example above, when computing points-to sets for `h`'s local

variables we conflate information for all of `c`'s contexts. This summarization effectively removes much of the fine detail about the behavior of `h` in different calling contexts. However, many applications require points-to information that is valid in every calling context. In addition, if we attempt to distinguish all call paths, the analysis can quickly become intractable.

6 Conclusion

We have explored two dimensions of the design space for flow-insensitive points-to analysis for C: polymorphic versus monomorphic and inclusion-based versus equality-based. Our experiments show that while polymorphism is potentially beneficial for equality-based points-to analysis, it does not have much benefit for inclusion-based points-to analysis. Even though we feel that added engineering effort can make the running times of the polymorphic analyses much faster, the precision would still be the same.

Monomorphic Andersen's analysis can be made fast [30] and often provides far more precise results than monomorphic Steensgaard's analysis. Polymorphic Steensgaard's analysis is in general much less precise than polymorphic Andersen's analysis, which is in turn little more precise than monomorphic Andersen's analysis. Additionally, as discussed in Sect. 4.3, implementing polymorphism is a complicated and difficult task. Thus, we feel that monomorphic Andersen's analysis may be the best choice among the four analyses.

Acknowledgements We thank the anonymous referees for their helpful comments. We would also like to thank Manuvir Das for suggestions for the implementation.

References

[1] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In X. Leroy and A. Ohori, editors, *Proceedings of the second International Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 78–96, Kyoto, Japan, Mar. 1998. Springer-Verlag.

[2] A. Aiken and E. L. Wimmers. Solving Systems of Set Constraints. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, June 1992.

[3] A. Aiken and E. L. Wimmers. Type Inclusion Constraints and Type Inference. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

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

[5] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Seventh Workshop on*

Languages and Compilers for Parallel Computing, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer-Verlag, 1994.

[6] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant Context Inference. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, San Antonio, Texas, Jan. 1999.

[7] M. Das. Unification-based Pointer Analysis with Directional Assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada, June 2000. To appear.

[8] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–24, San Diego, California, Jan. 1998.

[9] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, June 1994.

[10] N. Dor, M. Rodeh, and M. Sagiv. Detecting Memory Errors via Static Pointer Analysis. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34, Montreal, Canada, June 1998.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.

[12] M. Fähndrich. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, University of California, Berkeley, 1999.

[13] M. Fähndrich and A. Aiken. Program Analysis using Mixed Term and Set Constraints. In P. V. Hentenryck, editor, *Static Analysis, Fourth International Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126, Paris, France, Sept. 1997. Springer-Verlag.

[14] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[15] M. Fähndrich, J. Rehof, and M. Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada, June 2000. To appear.

[16] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. Technical Report UCB//CSD-97-964, University of California, Berkeley, Aug. 1997.

[17] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. Technical report, University of California, Berkeley, Apr. 2000.

[18] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, Philadelphia, Pennsylvania, June 1990.

[19] M. Hind and A. Pioli. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In G. Levi, editor, *Static Analysis, Fifth International Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy, Sept. 1998. Springer-Verlag.

[20] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, California, June 1992.

[21] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[22] C. Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[23] R. O'Callahan and D. Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, Massachusetts, May 1997.

[24] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. In B. Pierce, editor, *Proceedings of the 4th International Workshop on Foundations of Object-Oriented Languages*, Jan. 1997.

[25] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–88, Austin, Texas, Jan. 1989.

[26] E. Ruf. Context-Insensitive Alias Analysis Reconsidered. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, California, June 1995.

[27] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, San Antonio, Texas, Jan. 1999.

[28] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997.

[29] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, Jan. 1996.

[30] Z. Su, M. Fähndrich, and A. Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, Jan. 2000. To appear.

[31] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

[32] A. K. Wright. Simple Imperative Polymorphism. In *Lisp and Symbolic Computation 8*, volume 4, pages 343–356, 1995.

[33] S. H. Yong, S. Horwitz, and T. Reps. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, Georgia, May 1999.

[34] S. Zhang, B. G. Ryder, and W. A. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Fourth Symposium on the Foundations of Software Engineering*, Oct. 1996.

[35] S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with Combined Analysis for Pointer Aliasing. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 11–18, Montreal, Canada, June 1998.