Points-To for Java: A General Framework and an Empirical Comparison

Mirko Streckenbach, Gregor Snelting
Lehrstuhl für Softwaresysteme
Universität Passau
strecken@fmi.uni-passau.de

ABSTRACT

Points-to analysis for Java is different from points-to for C or even C++. We present a framework which generalizes popular points-to algorithms and generates set constraints from full Java bytecode. The framework exploits previously computed points-to sets in a fixpoint iteration for precise resolution of dynamic binding. We then compare implementations of this framework for unification-based and subset-based analysis. It turns out that – in contrast to the C situation – both approaches have about the same running time, while the subset-based algorithm is still more precise. The unification-based method is slowed down because its inherent imprecision accumulates during fixpoint iteration.

1. INTRODUCTION

Points-to analysis is a static analysis which computes for every pointer a set of objects it may point to at runtime. For imperative languages such as C, many points-to algorithms such as Andersen's algorithm [2], Steensgaard's algorithm [22], or Das' algorithm [7] have been investigated, and Steensgaard's algorithm is considered to be the fastest [14]. For object-oriented languages, dynamic binding must be approximated, and in fact for C++ dynamic binding can be analysed similar to function pointers in C [13].

For Java, however, the situation is different: there are no pointer arithmetics and no pointers to pointers, arrays have different semantics, type casts are type safe, and dynamic loading of classes is quite common. It is the aim of this article to generalize well-known points-to algorithms such as Andersen's and Steensgaard's (and some recent extensions) to full Java, exploiting Java's unique features. We will then provide an empirical comparison of these methods.

We begin with a small example illustrating that a careless treatment of dynamic binding will result in very unprecise points-to information. For the program in Figure 1, Andersen's algorithm will compute the initial points-to graph in Figure 1 (left). A naive treatment of method calls would then assume that in a call `x.f()`, all methods named `f()` from the static class of `x` and its subclasses can be targets of the call. It would then extend the points-to graph for any of these possible target methods by adding arcs which model implicit assignments to formal parameters, return values, and this-pointers. Thus for the example, all three implementations of method `f` will be included during the analysis of the calls to `f`, resulting in the final Andersen graph to be seen in Figure 1 (middle); the final Steensgaard graph for the same program can be seen in Figure 1 (right).[1]

The results are very unprecise – for example, `p` is assumed to point to three different objects, while it is obvious that `p` can only point to one object. Furthermore, the naive method assumes that in `a.f(c)` also `C.f` can be called, but in fact `a` cannot point to objects of type `C`, and in a type-safe language such as Java, the call to `C.f` can therefore be ignored. In fact, the final Steensgaard graph even contains type-incorrect edges such as `c`$\rightarrow$`new A`. Since pointer access along such edges will always generate an exception in Java, they can safely be ignored. Note that in C or C++, the latter consideration would not be valid.

The example demonstrates that for resolution of method calls, the actual points-to sets for the target object references should be considered. For a call `x.f()`, the points-to set for `x` gives possible target objects for the call, and static lookup for `f()` in the possible target objects will identify possible target methods for `f()`. This strategy will lead to a much smaller set of target methods than the naive method, which in turn increases precision since fewer arcs are added to the points-to graph. For the example, it leads to the points-to graphs in Figure 2. Still, the Steensgaard graph contains type-incorrect edges, due to its symmetric treatment of assignments.

In this contribution, we will elaborate on the insights from the above example. First we will present a generic framework for Java points-to analysis, which can be instantiated with subset-based intraprocedural approaches in the style of Andersen, or unification-based intraprocedural approaches in the style of Steensgaard. The framework consists of inference rules, which generate a constraint system for the points-to sets. It comprises the full Java language, and questions of library treatment and whole-program analysis are also

[1] For reasons of readability, this-pointers are not shown in Figure 1.

```
class MyException { Object u; MyException(Object t) { u=t } }
class A { A f(A g) { return g; }
class B extends A { A f(A g) { throw new MyException(g); } }
class C extends A { A f(A g) { return this; } }

...

A a=new A(), p=null, q, r, s;
B b=new B();
C c=new C();

if(...)
   a=b;

try {
   p=a.f(c);
} catch(MyException e) {
   q=e.u;
}

try {
   r=p.f(a);
} catch(MyException f) {
   s=f.u;
}
```

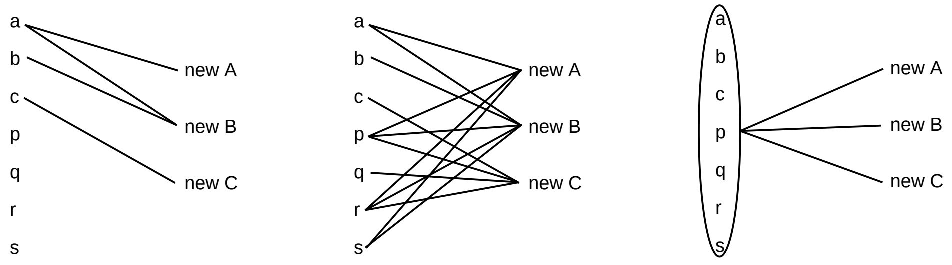

Figure 1: Example Java program and points-to graphs for naive treatment of dynamic binding. Left: initial Andersen graph, middle: final Andersen graph, right: final Steensgaard graph.

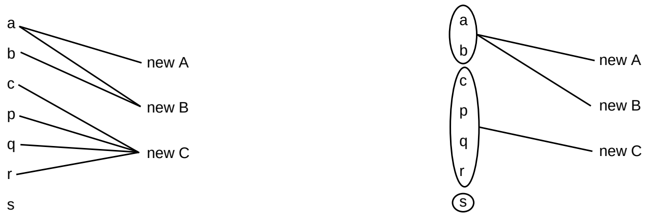

Figure 2: Final Andersen (left) and Steensgaard (right) graphs for improved resolution of dynamic binding.

discussed. For precise approximation of dynamic binding a fixpoint iteration is used, which exploits already computed points-to sets for a call's target object reference.

We then describe the implementation of specific instances of the framework. An empirical comparison will show that a subset-based approach is comparable in performance to a unification-based approach, but is of course more precise. The reason is that the imprecision in unification-based methods propagates during fixpoint iteration. The subset-based method can legitimately be called "Java-Andersen", while the unification-based method should *not* be called "Java-Steensgaard": it uses a Steensgaards-like approach for intraprocedural analysis, but uses fixpoint iteration for resolution of dynamic binding.

2. POINTS-TO FOR JAVA

Points to algorithms are usually described in terms of a points-to graph, which is a straightforward implementation of the points-to sets. Indeed, the implementation of our extended versions for Java is also based on points-to graphs. However, in order to make the idiosyncrasies of points-to for Java more clear, we will first present a formal description in terms of points-to sets and provide an inference system which generates constraints for the points-to sets (cmp. [11]). We use the following sets: Ptr, the set of all pointers (i.e. object references in Java); Obj, the set of all objects (i.e. constructor call sites);[2] Ass, the set of all assignments, where an assignment `l=r` is written as $(l,r) \in Ass$;[3] $Pt(p)$, the points-to set $\in 2^{Obj}$ for a pointer $p \in Ptr$.

The basic rules for intraprocedural points-to analysis are as follows. An assignment of an object reference to a pointer leads to inclusion of the object in the points-to set:

$$(p,o) \in Ass \wedge o \in Obj \Rightarrow o \in Pt(p)$$

Pointer assignments are treated differently by Andersen and Steensgaard. Andersen requires a subset relationship:

$$(p,q) \in Ass \wedge q \in Ptr \Rightarrow (Pt(q) \subseteq Pt(p))$$

Steensgaard merges the two points-to sets:

$$(p,q) \in Ass \wedge q \in Ptr \Rightarrow (Pt(q) = Pt(p))$$

The traditional algorithmic structure to solve the resulting constraint system can be seen in Figure 3: The constraints are generated in one pass over the program by collecting all explicit and implicit assignments; for Steensgaard the constraint system can then be solved in quasi-linear time, while Andersen requires an $O(n^3)$ iteration.

Shapiro and Horwitz[19] and Das[7] presented points-to algorithms which lie between Steensgaard and Anderson. Both extensions stick to general structure in Figure 3. For example, Das' algorithm can be adapted to Java by adding the following rule to the Andersen scheme:

$$(p,q) \in Ass \Rightarrow (Pt(p.f) = Pt(q.f))$$

where p, q are object references and f is a nested object reference.

[2]Different runtime incarnations for the same constructor call will – as usual – not be distinguished.
[3]Note that Ass also contains implicit assignments, such as assignments to parameters or this-pointers during method calls.

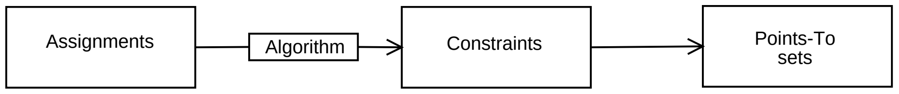

Figure 3: Structure of traditional points-to algorithms

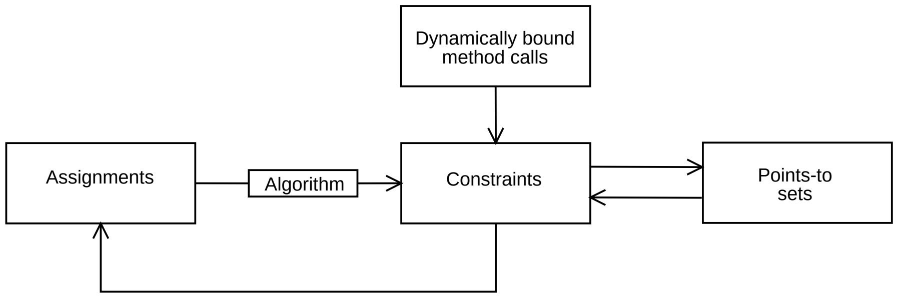

Figure 4: Structure of points-to algorithms for Java

Note that the Steensgaard rule can be modified for Java in order to avoid type-incorrect arcs in the points-to graph:

$$\begin{aligned}&(p,q) \in Ass \Rightarrow\\ &\quad\big((o \in Pt(q) \Rightarrow o \in Pt(p)\big) \wedge\\ &\quad\ \big(o \in Pt(p), type(o) \leq type(q) \Rightarrow o \in Pt(q)\big)\big)\end{aligned}$$

Interprocedural analysis simply uses the above rules in order to handle assignments to formal parameters, return values and this-pointers. In the presence of dynamic binding or function pointers, however, the situation becomes more complex. Steensgaard's algorithm can be extended for function pointers without using an additional fixpoint iteration [8]. However, Hind et al. [13] observed that the precise analysis of function pointers requires extensions of the basic mechanism. They proposed to use the points-to sets for function pointers in order to determine the possible call targets, and to extend the points-to graph according to these targets.

For Java, the analysis of method calls must be based on a similar approach: it must exploit the already computed points-to sets for the call's target object reference. For any possible target object in this set, the corresponding target method definition is determined according to the target object's static type. The call to the target method definition is then treated in the usual way by taking into account implicit assignments to formal parameters, return values, and this-pointers. Hence these assignments are valid only under the condition that a specific target object is in the points-to set. Therefore, the constraints for the resolution of dynamic binding have the following general form:

$$\big(o \in Pt(p) \wedge lookup(o,f) = C\big) \Rightarrow (\ldots,\ldots) \in Ass$$

where f is the called method; *lookup* determines the class which contains the appropriate definition of f according to the type of o. For the program in Figure 1, the call `p=a.f(c)` leads to three conditional constraints:

$$\begin{aligned}&\big(o \in Pt(\mathtt{a}) \wedge lookup(o,\mathtt{f}) = \mathtt{A}\big) \Rightarrow\\ &\quad\big((\mathtt{c},\mathtt{g}_\mathtt{A}) \in Ass \wedge (this^{\mathtt{f}}_{\mathtt{A}}, o) \in Ass \wedge (\mathtt{p},\mathtt{g}_\mathtt{A}) \in Ass\big)\end{aligned}$$

$$\begin{aligned}&\big(o \in Pt(\mathtt{a}) \wedge lookup(o,\mathtt{f}) = \mathtt{B}\big) \Rightarrow\\ &\quad\big((\mathtt{c},\mathtt{g}_\mathtt{B}) \in Ass \wedge (this^{\mathtt{f}}_{\mathtt{B}}, o) \in Ass\big)\end{aligned}$$

$$\begin{aligned}&\big(o \in Pt(\mathtt{a}) \wedge lookup(o,\mathtt{f}) = \mathtt{C}\big) \Rightarrow\\ &\quad\big((\mathtt{c},\mathtt{g}_\mathtt{C}) \in Ass \wedge (this^{\mathtt{f}}_{\mathtt{C}}, o) \in Ass \wedge (\mathtt{p}, this^{\mathtt{f}}_{\mathtt{C}}) \in Ass\big)\end{aligned}$$

The assignments generated by these rules will then generate additional set constraints according to the Andersen resp. Steensgaard rule. Note that the explicit generation of assignments decouples interprocedural analysis from the choice of the intraprocedural algorithm. Note further that the naive method will generate the same assignments, but without the conditions; hence it would generate 8 assignments as compared to 2 or 3. In general, conditional constraints will never generate more assignments (and hence points-to relations) than the naive method, but usually much less.

The correctness of the above constraints is obvious. But in order to solve such a system of conditional constraints, an additional feedback loop is needed (see Figure 4): points-to entries can generate new constraints, which can then extend the set of assignments. The algorithm structure is as follows:

```
do {
  apply basic algorithm;
  evaluate conditional constraints;
} while (points-to graph changed)
```

3. GENERATING CONSTRAINTS FROM JAVA BYTECODE

Java Bytecode is more stable than the Java source language, and many programs are available as Bytecode but not in source form. We therefore present the details of the constraint-generating inference system for Bytecode. Due to space limitations, we will present only a few central rules; the full inference system – including in particular static calls and exceptions – can be found in [23]. The general structure of the inference rules is as follows:

$$\frac{\text{Bytecode} \mid \text{old stack}}{\text{constraints} \mid \text{types} \mid \text{new stack}}$$

Rules are applied to bytecode instructions in sequential order, thereby generating constraints and some auxiliary information. The premises of a rule match a specific bytecode instruction. Bytecode instructions refer to stack elements, thus an abstraction of the JVM stack contents is used in the rules as well. For runtime stack values, their abstract representation is the corresponding variable name, which can in most cases be extracted from the compiler's variable table.[4]

The conclusion of each inference rule contains in its left part the constraints generated from the bytecode instruction. Furthermore, some typings for pointers are reconstructed. The last part of the conclusion displays the modified abstract stack as to be used for the next bytecode instruction in its matching rule premise. For better readability, conditional constraints are split into two rules: since type and stack information do not depend on points-to information, they are purely static and are factored out in a separate rule.

The inference rules for constraint generation are presented in Figure 5. As a typical example, consider the rule for virtual method call. The premise of the static part makes assumptions about the bytecode instruction and the signature of the method in question. In addition, the premise names the (abstractions of the) actual parameters on the stack. The conclusion of the static part states that the this-pointer and the return value are indeed pointers, and names their type. The next bytecode must be matched against an inference rule using a modified abstract stack, where the parameter abstractions have disappeared and are replaced by the abstract return value.

Of course, the interesting part is the dynamic part, which generates assignments under the assumption that some object is in the points-to set for the call's object reference. The latter reference (more precisely, its abstract form, that is a variable name) is taken from the abstract stack (see static part). The premise of the dynamic part determines the corresponding method definition by static lookup. The conclusion generates one asignment for every formal/actual parameter pair, for the method's this-pointer, and for its return value. It also gives types for the callee's this-pointer.

Let us apply the rules to a small program fragment and its bytecode (Figure 6). The abstract stack as well as the generated assignments can be seen in the lower part of Figure 6. Application of the rule for the first bytecode instruction results in two statements: the object, which is created in method $f(int)$ from class S at bytecode address 0 is indeed an object, and has type A. The this-pointer of the default constructor method is initialised by the program, hence a corresponding unconditional assignment is generated. The next assignment, corresponding to the initialisation of variable `a`, is also unconditional. Of course, the interesting part is the `invokevirtual` instruction and its corresponding abstract parameter entry on the stack. The application of the `invokevirtual`-rule generates two condititional constraints, each consisting of three assignments and some additional type information. Note how the two constraints mirror the two possibilities for dynamic binding of method `f`: it could be `A.f` or `B.f`. Finally, the `putfield` instruction generates an unconditional assignment for `v`, and a conditional assignment for any object which might be pointed to by `f`'s this-pointer (since it will also contain field `v`).

Note that the example program contains a conditional expression, which generates a stack entry whose abstract version cannot be taken from the variable table; instead the possible control flows from method entry to the call of f must be explored. This results in two abstract top stack entries, namely `S.f(int).<#18>` and `S.f(int).<#e>` (that is, the `new B` resp. `new C` construction site). In the following, we explore only the first alternative.[5] The initial constraints for the points-to sets, according to Andersen's algorithm, are as follows:

$\texttt{S.f(int).<\#0>} \in Pt(\texttt{S.f(int).a})$
$\texttt{S.f(int).<\#0>} \in Pt(\texttt{A.<init>().<this>})$
$\texttt{S.f(int).<\#18>} \in Pt(\texttt{C.<init>().<this>})$
$Pt(\texttt{A.f(Object->Object).<return>})$
$\quad \subseteq Pt(\texttt{S.f(int).<this>.(S.o)})$

After one iteration, the final results are obtained:

$\texttt{S.f(int).<\#0>} \in Pt(\texttt{S.f(int).a})$
$\texttt{S.f(int).<\#0>} \in Pt(\texttt{A.<init>().<this>})$
$\texttt{S.f(int).<\#18>} \in Pt(\texttt{C.<init>().<this>})$
$Pt(\texttt{A.f(Object->Object).<return>})$
$\quad \subseteq Pt(\texttt{S.f(int).<this>.(S.o)})$
$\texttt{S.f(int).<\#0>} \in Pt(\texttt{A.f(Object->Object).<this>})$
$\texttt{S.f(int).<\#18>} \in Pt(\texttt{A.f(Object->Object).p})$
$Pt(\texttt{A.f(Object->Object).<return>})$
$\quad \subseteq Pt(\texttt{A.f(Object->Object).<return/S.f(int).a>})$

The solution shows that `a` as well as `f`'s this-pointer can only point to an `A` object; hence the call `a.f(...)` has only `A.f` as a target method. Both the naive method as well as call-graph based methods such as Rapid Type Analysis [3] would be unable to exclude `B.f` as a possible target.

4. WHOLE-PROGRAM-ANALYSIS, NATIVE CODE AND REFLECTION

Many programs use library functions for which there is no source text available, or which are not written in Java ("unanalysed functions"). A popular way to deal with this situation is to provide stubs for these functions, that is source code fragments which simulate the points-to behaviour of the function. However, the effort for stub implementation and maintenance is enormous.

An alternative is to use a conservative approximation for unknown bytecode. Unknown functions can do anything with

[4]In pathological examples, the reconstruction of abstract stack values can lead to combinatorial explosion due to an exponential number of control flow paths between two program points. But in practice, this never happens. An alternative to get rid of this phenomenon alltogether is to analyse source code instead of Bytecode.

[5]In the rare case of non-unique abstract stack entries, several variants of the constraint system will be generated and solved.

Object creation:
$$\frac{\begin{array}{c|c} I \equiv \texttt{new}\ A & S = [\ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} m.\langle adr(I)\rangle \in Obj \\ type(m.\langle adr(I)\rangle) = A \end{array} & S = [m.\langle adr(I)\rangle, \ldots] \end{array}}$$

Assignment:
$$\frac{\begin{array}{c|c} I \equiv \texttt{astore}\ r & S = [p, \ldots] \end{array}}{\begin{array}{c|c|c} (Register(m, r, b_{i+1}), p) \in Ass & Register(m, r, b_{i+1}) \in Ptr & S = [\ldots] \end{array}}$$

Virtual call (static part):
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{invokevirtual}\ m \\ sig(m) = (t_1, \ldots, t_n) \to t \end{array} & S = [p_r \ldots p_1, q, \ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} m.\langle \texttt{ret}/q\rangle \in Ptr \\ type(m.\langle \texttt{ret}/q\rangle) = t \end{array} & S = [m.\langle \texttt{ret}/q\rangle, \ldots] \end{array}}$$

Virtual call (conditional constraints):
$$o \in Pt(q) \Rightarrow \frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{invokevirtual}\ m \\ sig(m) = (t_1, \ldots, t_n) \to t \\ m' = lookup(o, m) \end{array} & \end{array}}{\begin{array}{c|c|c} \begin{array}{c} \forall_{i=1}^{n}(par(m', i), p_i) \in Ass, \\ (m'.\langle \texttt{this}\rangle, q) \in Ass, \\ (m.\langle \texttt{ret}/q\rangle, m'.\langle \texttt{ret}\rangle) \in Ass \end{array} & \begin{array}{c} m'.\langle \texttt{this}\rangle \in Ptr \\ type(m'.\langle \texttt{this}\rangle) = cls(m') \\ m'.\langle \texttt{ret}\rangle \in Ptr \\ type(m'.\langle \texttt{ret}\rangle) = t \end{array} & \end{array}}$$

Data member access (static part):
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{getfield}\ f \\ f' = LookupField(f) \end{array} & S = [p, \ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} p.f' \in Ptr \\ type(p.f') = type(f') \end{array} & S = [p.f', \ldots] \end{array}}$$

Data member access (conditional constraints): $o \in Pt(p) \Rightarrow$
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{getfield}\ f \\ f' = LookupField(f) \end{array} & \end{array}}{\begin{array}{c|c|c} (p.f', o.f') \in Ass & \begin{array}{c} o.f' \in Ptr \\ type(o.f') = type(f') \end{array} & \end{array}}$$

Data member store (static part):
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{getfield}\ f \\ f' = LookupField(f) \end{array} & S = [p, \ldots] \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} p.f' \in Ptr \\ type(p.f') = type(f') \end{array} & S = [p.f', \ldots] \end{array}}$$

Data member store (conditional constraints): $o \in Pt(p) \Rightarrow$
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{getfield}\ f \\ f' = LookupField(f) \end{array} & \end{array}}{\begin{array}{c|c|c} (p.f', o.f') \in Ass & \begin{array}{c} o.f' \in Ptr \\ type(o.f') = type(f') \end{array} & \end{array}}$$

Array element store (static part):
$$\frac{\begin{array}{c|c} I \equiv \texttt{aastore}\ f & \begin{array}{c} S = [v, c, p, \ldots] \\ type(p) = t[\ ] \end{array} \end{array}}{\begin{array}{c|c|c} (p[\ ], v) \in Ass & \begin{array}{c} p[\ ] \in Ptr \\ type(p[\ ]) = t \end{array} & S = [\ldots] \end{array}}$$

Array element store (conditional constraint): $o \in Pt(p) \Rightarrow$
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{aastore}\ f \\ type(o) = t[\ ] \\ o[\ ] \leq type(v) \end{array} & \end{array}}{\begin{array}{c|c|c} (o[\ ], v) \in Ass & \begin{array}{c} o[\ ] \in Ptr \\ type(o[\ ]) = t \end{array} & \end{array}}$$

Type cast (static part):
$$\frac{\begin{array}{c|c} I \equiv \texttt{checkcast}\ t & \begin{array}{c} S = [p, \ldots] \\ \neg(p = null \vee type(p) \leq t) \end{array} \end{array}}{\begin{array}{c|c|c} & \begin{array}{c} (t)p \in Ptr \\ type((t)p) = t \end{array} & S = [(t)p, \ldots] \end{array}}$$

Type cast (conditional constraint): $o \in Pt(p) \Rightarrow$
$$\frac{\begin{array}{c|c} \begin{array}{c} I \equiv \texttt{checkcast}\ t \\ type(o) \leq t \end{array} & \end{array}}{\begin{array}{c|c|c} ((t)p, o) \in Ass & & \end{array}}$$

Figure 5: Constraint-generating rules for some Bytecode instructions

```
class A {
   Object f(Object p) { ... }
}
class B extends A {
   Object f(Object q) { ... }
}
class C {}

class S {
   Object v;

   void f(int x) {
      A a=new A();
      v=a.f(x>0?(Object)new B():(Object)new C());
   }
}
```

```
#0:    new A
       dup
       invokespecial A.<init>()
       astore_2
       aload_0
       aload_2
       iload_1
       ifle #18 -> #e #18
#e:    new B
       dup
       invokespecial B.<init>()
       goto #1f -> #1f
#18:   new C
       dup
       invokespecial C.<init>()
       invokevirtual A.f(java.lang.Object->java.lang.Object)
       putfield S.v
       return -> end
```

	Instruction	Stack after rule application	Auxiliary Information
#0	new A	S.f(int).<#0>	S.f(int).<#0> ∈ Obj, typ(S.f(int).<#0>)=A
#3	dup	S.f(int).<#0> S.f(int).<#0>	
#4	invokespecial A.<init>()	S.f(int).<#0>	(A.<init>().<this>,S.f(int).<#0>) ∈ Ass
#7	astore_2		(S.f(int).a,S.f(int).<#0>) ∈ Ass
#8	aload_0	S.f(int).<this>	
#9	aload_2	S.f(int).a S.f(int).<this>	
#a	iload_1	S.f(int).x/int S.f(int).a S.f(int).<this>	
#b	ifle #18	S.f(int).a S.f(int).<this>	
#18	new C	S.f(int).<#18> S.f(int).a S.f(int).<this>	S.f(int).<#18> ∈ *Obj*, typ(S.f(int).<#18>)=C
#1b	dup	S.f(int).<#18> S.f(int).<#18> S.f(int).a S.f(int).<this>	
#1c	invokespecial C.<init>()	S.f(int).<#18> S.f(int).a S.f(int).<this>	(C.<init>().<this>,S.f(int).<#18>) ∈ Ass
#1f	invokevirtual A.f(Object->Object)	A.f(Object->Object).<return/S.f(int).a>	typ(A.f(Object->Object).<return/S.f(int).a>)=Object A.f(Object->Object).<return> ∈ Ptr, typ(A.f(Object->Object).<return>) = Object *o* ∈ Pt(S.f(int).a)∧ LookupVirtual(*o*,A.f(Object->Object)) = A.f(Object->Object) ⇒ A.f(Object->Object).<this> ∈ Ptr, typ(A.f(Object->Object).<this>)=A ∧ (A.f(Object->Object).<this>,o) ∈ Ass ∧ (A.f(Object->Object).p,S.f(int).<#18>) ∈ Ass ∧ (A.f(Object->Object).<return/S.f(int).a>,A.f(Object->Object).<return>) ∈ Ass *o* ∈ Pt(S.f(int).a)∧ LookupVirtual(*o*,A.f(Object->Object)) = B.f(Object->Object) ⇒ B.f(Object->Object).<this> ∈ Ptr, typ(B.f(Object->Object).<this>)=A ∧ (B.f(Object->Object).<this>,o) ∈ Ass ∧ (B.f(Object->Object).q,S.f(int).<#18>) ∈ Ass ∧ (A.f(Object->Object).<return/S.f(int).a>,A.f(Object->Object).<return>) ∈ Ass
#22	putfield S.v		S.f(int).<this>.(S.v) ∈ Ptr, typ(S.f(int).<this>.(S.v))=Object (S.f(int).<this>.(S.v),A.f(Object->Object).<return>) ∈ Ass *o* ∈ Pt(S.f(int).<this>) ⇒ (o.(S.v),A.f(Object->Object).<return>) ∈ Ass o.(S.v) ∈ *Ptr*, typ(o.(S.v)) = Object
#25	return		

Figure 6: A program fragment, its Bytecode, and the corresponding application of constraint-generating rules.

$$\frac{m \text{ is unanalysed},\quad sig(m) = (t_1, \ldots, t_n) \rightarrow t}{\forall_{i=1}^{n} (unanalysed, m.{<}par(m,i){>}) \in Ass}$$

$$\frac{m \text{ is unanalysed, } sig(m) = (t_1, \ldots, t_n) \rightarrow t}{(m'.{<}\mathtt{ret}{>}, unanalysed_t) \in Ass}$$

$$o \in Pt(unanalysed/t) \Rightarrow \quad \frac{\begin{array}{c} m \text{ is method in } t, \\ m \text{ is visible in unanalysed code}, \\ sig(m) = (t_1, \ldots, t_n) \rightarrow t \\ m' = lookup(o, m'), \end{array}}{\begin{array}{c} \forall_{i=1}^{n} (par(m',i), unanalysed_{t_i}) \in Ass, \\ (m'.{<}\mathtt{this}{>}, o) \in Ass, \\ (unanalysed, m'.{<}\mathtt{ret}{>}) \in Ass \end{array}}$$

Figure 7: Constraint generation for unanalysed functions

their parameters; in particular any object given to an unanalysed function may reappear as the return value of any other unanalysed function. But Java's type-safety can be exploited to deliver some precision even under this conservative and precision-threatening assumption.

In order to deal with unanalysed code, we first introduce a global variable "*unanalysed*". Whenever an object reference p is passed to an unanalyzed function, an assignment $(unanalysed, p)$ is added. Pointers which are returned from unanalysed functions could analogeously modelled as assignments $(p, unanalysed)$. But for return values at least a type is known and should be utilized for increased precision. We thus introduce special versions of *unanalysed*, namely global variables $unanalysed_t$ for every type t. The relation between *unanalysed* and the various $unanalysed_t$ is given by

$$o \in Pt(unanalysed) \wedge type(o) \leq t \Rightarrow o \in Pt(unanalysed_t)$$

Instead of inserting the assignment $(p, unanalysed)$ whenever the return value of an unanalysed function is assigned to p, we insert $(p, unanalysed_t)$, which reduces the size of p's points-to set. Note that this "trick" can only be done in a type-safe language as Java, but not in C or C++.

Figure 7 (upper part) gives the inference rules which generate the corresponding assignments. Unanalysed functions can call other functions (unanalysed or analysed), and the rule in Figure 7 (lower part) describes such calls. It is similar to the `invokevirtual` rule, except that there is no stack, and parameters and return values of functions called from unanalysed functions are again modelled via the *unanalysed* variable. Exceptions and access to global variables from unanalysed code can be modelled similarly; for details see [23].

Let us conclude this section with a discussion of the reflection API. Native methods from the reflection API could of course be analysed using the above approximations, but in many cases, we can do better. As an example, consider the calls to `getClass` in Figure 8. For every class t in the program, we introduce a special object $class_t$. In order to analyse the call `c=a.getClass();`, we first determine $Pt(\texttt{a})$. The types of all objects in this set determine which $class_t$ have to be added to $Pt(\texttt{c})$. In the example, $Pt(\texttt{c}) = \{class_A\}$. Therefore the call `o=c.newInstance():` will return a new special object $dyn : a$ of type A. As a result, the call `a2.f();` can be resolved.

This approach is more precise than traditional stubs, because again it incorporates points-to information for target objects and parameters. Providing specific constraints for some popular unanalysed functions improves precision considerably, while the above general approximation can be used for less popular unanalysed functions without harming precision too much.

5. EMPIRICAL STUDIES

We implemented the framework as well as its subset-based and unification-based variants (see [23] for implementation details). The implementation is based on points-to graphs. For Andersen's method, every assignment, let it be static or conditionally generated, results in an additional arc in the graph.[6] In order to reduce memory consumption, the implementation does not store complete points-to sets, but allows transitive edges in the graph; complete points-to sets are then determined by traversion of paths in the graph.

Instead of adding edges, the implementation of unification-based intraprocedural analysis merges graph nodes via the fast union-find algorithm. We have already seen that this can introduce type-incorrect points-to relations. Therefore, additional type checks are performed whenever a points-to set is explicitely needed.

We applied both variants to 22 small and medium-sized programs with up to 25000 LOC. The results are summarized in Figure 9. The first columns give the program name, its number of classes, number of methods, Bytecode size, and number of calls. Furthermore the percentage of calls which could be resolved statically even without points-to information is given. It is interesting to see that this percentage is usually well above 80% – Java programs rely heavily on the standard API, which contains many `final` methods. Of course, statically resolvable method calls do not need conditional constraint generation, but can be analysed directly.

For both methods, the following data are given: runtime, relative precision in percent, percentage of additionally resolved method calls, and some information concerning our specific application of points-to analysis. The relative precision is determined in comparison with a super-naive points-to method, where every points-to set contains all objects which have a correct type:

$$Pt_{SN}(p) = \{o \in Obj \mid type(o) \leq type(p)\}$$

This method is even worse than the naive method from the introduction, because not only it resolves method calls in a naive way, it even makes very imprecise assumptions for intraprocedural analysis. Relative precision is defined as

$$RP = \frac{\sum_{p \in Ptr} |Pt(p)|}{\sum_{p \in Ptr} |Pt_{SN}(p)|}$$

An algorithm with relative precision less than 1 (or below 100%) is thus better than the super-naive method.

The runtimes have been determined on a SUN Enterprise system 450 with 1GB, running JDK1.2. Looking at the runtimes, there are two basic observations. First of all, the absolute runtimes are quite high. The reason, of course, is that precise analysis of dynamic binding does not come for free. One might imagine a better implementation, or the use of a dedicated, highly optimized constraint solver, but fact is that precise resolution of dynamic binding requires an additional level of fixpoint iteration. Furthermore, unanalysed functions often induce quite conservative assumptions which reduce precision and speed of the analysis.

The second fundamental observation is that the runtimes are relatively similar. The sum of all runtimes in the bench-

[6]The basic Andersen rule can be expressed solely in assignments: $\big((p,q) \in Ass \Rightarrow (Pt(q) \subseteq Pt(p))\big) \iff$ $\big((p,q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow o \in Pt(p))\big) \iff$ $\big((p,q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow (p,o) \in Ass)\big)$. Similar for Das and Steensgaard.

```
class A {
   void f() { ... }
}

class B extends A {
   void f() { ... }
}

class Main {
   void main() throws IllegalAccessException, InstantiationException {
      A a=new A();
      Class c=a.getClass();
      Object o=c.newInstance();
      A a2=(A)o;
      a2.f();
   }
}
```

Figure 8: Example use of reflection API

						subset-based						unification-based				
Program	Cl.	Me.	code	calls	static	time	RP	scc	res.	client	client-t.	time	RP	res.	client	client-t.
Haar	17	230	23k	1011	94.0	51.53	71.7	0 / 3	5.9	16.2	13.01	59.98	95.0	4.3	16.9	13.73
IComputer	63	390	39k	2261	94.2	162.31	58.9	14 / 14	5.1	22.8	55.91	282.21	84.6	4.9	22.9	59.60
JBinHex	5	54	3024	75	81.3	7.12	77.7	2 / 2	18.7	33.2	1.11	5.68	96.8	6.7	33.4	1.14
JLex	26	161	28k	1063	97.7	45.24	67.4	7 / 13	1.9	20.0	8.96	75.72	94.1	1.9	20.2	8.87
Jccodes	8	41	2897	158	68.4	19.98	74.7	2 / 2	21.5	23.5	3.05	7.40	95.7	21.5	23.5	3.07
NanoXML	3	32	2296	158	96.2	6.64	81.0	2 / 2	3.8	16.0	1.09	4.89	98.4	3.8	16.0	1.12
ProxyHammer	12	38	3079	242	88.8	14.76	73.9	0 / 0	10.7	31.8	2.74	11.02	98.5	10.7	31.8	2.68
TextScroll	4	92	6644	425	73.2	16.71	68.0	0 / 0	26.8	6.4	4.35	14.02	95.6	26.8	6.4	4.39
TumblingDice	34	196	12k	762	90.8	50.12	73.6	2 / 2	8.8	17.7	13.37	47.10	97.0	8.8	17.7	13.13
arabeske	21	296	41k	1964	82.9	162.75	76.6	3 / 5	4.3	16.5	18.85	163.21	94.7	4.3	16.5	19.18
graph	32	228	16k	1192	96.7	47.98	66.9	1 / 3	3.2	24.2	14.78	71.48	96.4	3.2	24.2	14.99
hanoi	45	362	21k	1005	82.5	43.96	72.2	1 / 18	13.5	20.1	17.37	44.81	95.2	13.0	20.7	14.78
j6502	1	31	8123	78	100.0	3.21	40.3	0 / 0	0.0	10.7	0.48	1.73	98.0	0.0	10.7	0.49
jEdit	108	489	34k	2179	81.1	430.12	70.7	23 / 33	13.9	20.3	117.15	473.89	88.6	13.2	20.3	115.39
jas	127	435	26k	1042	85.1	611.56	84.5	12 / 24	3.7	4.9	54.18	404.69	99.6	3.7	4.9	54.10
java_cup	41	396	32k	2362	93.2	142.71	71.6	5 / 11	3.4	19.5	35.58	301.58	97.4	3.4	19.5	33.54
jaxp	110	761	39k	1579	78.3	528.30	84.1	27 / 93	7.6	11.9	138.74	596.49	96.8	6.2	12.0	134.03
jflex	52	418	50k	2266	96.9	196.77	81.3	4 / 11	2.6	17.6	42.24	373.38	95.5	2.5	17.6	38.53
jspringies	13	71	8045	239	90.8	15.19	68.3	2 / 2	8.4	15.6	3.17	7.68	97.4	8.4	15.6	2.98
mars	19	120	5431	371	92.7	39.04	64.4	2 / 4	2.4	18.1	8.51	28.02	92.4	2.4	18.3	8.42
sablecc	283	1867	74k	4562	68.2	949.04	62.3	2 / 72	20.1	4.0	363.71	1069.11	88.5	17.7	4.0	343.45
yamm	71	264	39k	3279	89.8	1270.15	58.5	5 / 6	7.1	24.5	93.62	1390.87	78.5	6.2	24.5	91.84

Figure 9: Benchmark results

mark is 4815.22 seconds for the subset-based, 5434.96 for the unification-based version; that is a difference of roughly 10 percent. Obviously the iteration for dynamic binding destroys the basic speed of the unification-based method.

Concerning relative precision, the subset-based method is on the average 32.8% more precise than the super-naive method, and the unification-based method is on the average 5.7% more precise than the super-naive method. This is a disappointing result for the symmetric unification-based approach; indicating that it is unsuitable for the abundance of unsymmetric subtype relations in Java programs.

The difference is less significant if we look only at the number of statically resolved method calls. The columns "res." give the percentage of calls which could not be resolved statically, but where the points-to set is so small that the target method is unique. Again, the subset-based method is better. Adding the values in column "static" and in column "res.", both methods achieve almost 100% for most programs. That is, dynamic binding is hardly used in the benchmark. Comparing this with the relative precision, the reader should keep in mind that the majority of pointers is not used as target objects for method calls.

We also incorporated an algorithm for strongly connected components as described in [17]. Rountev reports very positive effects for C programs, but for Java, the results are disappointing. The column "scc" presents the number of strongly connected components in the Andersen graph before and after fixpoint iteration. Both numbers are so low that there is no improvement in practice. Again, we believe that the unsymmetric subtype relations which are so typical for OO programming prevent the approach from being effective in Java.

Let us finally consider the effect of the two different methods on a specific client analysis, namely the KABA system as described in [21, 20]. KABA starts out with a table containing all method accesses for every program variable, and in order to compute the table, points-to information is needed for every program variable. The better the points-to analysis, the less non-blank table entries. The columns labelled "client" display the percentage of table entries which are not blank, and the columns labelled "client-t" give table construction time. Similarly to the "resolved calls" results, the subset method is only slightly superior to the other one for this specific client analysis.

6. RELATED WORK

Rountev, Milanova, and Ryder recently presented the only other implemented points-to algorithm for Java known to us [18]. Their method is also based on set constraints, but is limited to Andersen's approach; they do not consider approximations for unanalysed code and the reflection API. Rountev et al. use Soot[7] as a frontend and the BANE system [1] for solving set constraints. The implementation has roughly the same speed as ours, but uses less memory. This is probably due to their use of the highly optimized BANE engine (see [10, 24]).

It would be interesting to compare the precision of the two systems, but right now this is not possible: Rountev et al. analyse reachable methods in user and library code; we analyse the whole user code and treat libraries as unanalysed code. Furthermore, the programs common to both benchmarks are obviously not the same version. In any case, a comparison not only of the resolved calls but also of the relative precision would be worthwhile.[8]

Recently, Steensgaard's algorithm has been extended to Java as well [8]. In contrast to our unification-based variant, it does not use fixpoint iteration, but – in case two variables a and b have been unified – unifies the signatures and this-pointers of all methods in a's and b's static type. This retains the quasi-linear speed of the method, but is less precise than our approach. [8] reports that a reasonable precision can only be achieved if a context-sensitive extension is used.

7. CONCLUSION AND FUTURE WORK

We presented a comparison of a subset-based and a unification-based points-to approach for Java. Our results can be summarized as follows:

1. Both analysis strategies differ only in one specific inference rule, which is plugged into a generic points-to framework for Java.
2. Java's type safety can be exploited to increase precision, in particular for unanalysed code.
3. Unification-based methods have difficulties with the abundancy of unsymmetric subtype relations in Java Programs.

While in the world of imperative languages such as C, Steensgaard's method is much faster than Andersen's, intraprocedural Steensgaard combined with fixpoint iteration for dynamic binding is slightly slower for Java. The reason is that the fixpoint iteration leads to a propagation of the imprecision in Steensgaard's method, and eventually to slower convergence.

There is still much room for improvement, both in precision and performance. Besides better implementations of our algorithms, two options seem worth exploring: context-sensitive points-to analysis and flow-sensitive points-to analysis. A partially flow-sensitive analysis can easily be achieved by transforming the Bytecode to static single assignment form (cmp. [12]). Context-sensitive points-to analysis for Java can be achieved in a way analogeously to [15], [9] or [5]. This will increase precision, but it is unclear how high the price will be in terms of performance, and what the performance/precision ratio will be.

Our analysis is basically a whole-program analysis. But it is known that many Java objects never leave the methods which have created them [4, 6, 25]. It should thus be possible to deal with local pointers and objects at the level of methods, thereby decreasing the size of the global points to graph. For C, this has alrady been done [16]; for Java, it remains to be seen whether it is possible.

[7]`http://www.sable.mcgill.ca`

[8]Our current implementation requires that the bytecode comes with debug information.

8. REFERENCES

[1] A. Aiken, M. Faehndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. *Lecture Notes in Computer Science*, 1473:78–92, 1998.

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

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341. ACM Press, 1996.

[4] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[5] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proc. 26th ACM SIGPLAN-SIGACT on Principles of programming languages*, ACM SIGPLAN Notices, pages 133–146, New York, NY, USA, 1999. ACM Press.

[6] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, October 1999.

[7] Manuvir Das. Unification-baseb pointer analysis with directional assignments. In *Proc. SIGPLAN Conference on Programmming Design and Implementation (PLDI)*, pages 35–46, Vancouver, Canada, June 2000.

[8] Manuvir Das and Bjarne Steensgaard, November 2000. Personal communication.

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256. ACM, ACM, June 1994.

[10] Manuel Fähndrich, Jeffrey Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998. *ACM SIGPLAN Notices* 33(6).

[11] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley, August 5, 1997.

[12] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 97–105, Montreal, Canada, 17–19 June 1998.

[13] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[14] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proc. International Symposium on Software Testing and Analysis*, pages 113–123, Portland, OR, 2000.

[15] William Landi and Barbara G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1992.

[16] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. ESEC/FSE*, pages 199–215, N. Y., September 6–10 1999. ACM Press.

[17] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programmming Design and Implementation (PLDI)*, pages 47–56, Vancouver, Canada, June 2000.

[18] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated inclusion constraints. Technical Report DCS-TR-417, Department of Computer Science, Rutgers University, July 2000.

[19] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, New York, NY, USA, 1997. ACM Press.

[20] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems.* to appear.

[21] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering: FSE-6*, pages 99–110. ACM Press, 1998.

[22] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, January 1996.

[23] M. Streckenbach. Points-to-Analyse für Java. Number MIP-0011 in Technical Report Series. Fakultät für Mathematik und Informatik, Universität Passau, 2000.

[24] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95, Boston, Massachusetts, January 19–21, 2000.

[25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, October 1999.