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 [11] 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).

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 -> 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

1For reasons of readability, this-pointers are not shown in Figure 1.
A = new A(); p = null, q, r, s;
B = new B();
C = new C();

if(...)
a = b;
try {
p = 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 naïve 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 meth-
ods 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 Steensgaard-like approach for in-
traprocedural analysis, but uses fixpoint iteration for reso-
lution 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 ex-
tended 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 descrip-
tion in terms of points-to sets and provide an inference sys-
tem which generates constraints for the points-to sets (cmp.
[11]). We use the following sets: $\text{Ptr}$, the set of all pointers
(i.e. object references in Java); $\text{Obj}$, the set of all objects
(i.e. constructor call sites); $\text{Ass}$, the set of all assignments,
where an assignment $l = r$ is written as $(l, r) \in \text{Ass}$; $\text{Ptr}$, the
points-to set $\in 2^{\text{Obj}}$ for a pointer $p \in \text{Ptr}$.

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:

$(p, o) \in \text{Ass} \land o \in \text{Obj} \Rightarrow o \in \text{Ptr}(p)$

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

$(p, q) \in \text{Ass} \land q \in \text{Ptr} \Rightarrow (\text{Ptr}(q) \subseteq \text{Ptr}(p))$

Steensgaard merges the two points-to sets:

$(p, q) \in \text{Ass} \land q \in \text{Ptr} \Rightarrow (\text{Ptr}(q) = \text{Ptr}(p))$

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 al-
gorithms which lie between Steensgaard and Andersen. Both
extensions stick to general structure in Figure 3. For exam-
ple, Das’ algorithm can be adapted to Java by adding the
following rule to the Andersen scheme:

$(p, q) \in \text{Ass} \Rightarrow (\text{Ptr}(p.f) = \text{Ptr}(q.f))$

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

2Different runtime incarnations for the same constructor call
will – as usual – not be distinguished.

3Note that $\text{Ass}$ also contains implicit constructor calls, such as
assignments to parameters or this-pointers during method

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

\[
(p, q) \in \text{Ass} \Rightarrow \\
((o \in \text{Pt}(q) \Rightarrow o \in \text{Pt}(p)) \land \\
(o \in \text{Pt}(p), \text{type}(o) \leq \text{type}(q) \Rightarrow o \in \text{Pt}(q))
\]

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:

\[
(o \in \text{Pt}(p) \land \text{lookup}(o, f) = C) \Rightarrow (\ldots, \ldots) \in \text{Ass}
\]

where \( f \) is the called method; \( \text{lookup} \) 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:

\[
(o \in \text{Pt}(a) \land \text{lookup}(o, f) = A) \Rightarrow \\
((c.g_A) \in \text{Ass} \land (\text{this}_A, o) \in \text{Ass}) \land (p.g_A) \in \text{Ass}
\]

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:

\[
do \{
\ \ \ \text{apply basic algorithm;}
\ \ \ \text{evaluate conditional constraints;}
\}\ \text{while (points-to graph changed)}
\]

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:
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 (abstracts 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 assignment 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 conditional 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. \(C\) 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:

\[
\begin{align*}
S.f(int), \#18 &\in P(S.f(int).a) \\
S.f(int), \#e &\in P(A.f(int), \langle \text{this} \rangle) \\
S.f(int), \#18 &\in P(C, \langle \text{init} \rangle, \langle \text{this} \rangle) \\
P(A.f(Object->Object), \langle \text{return} \rangle) \\
&\subseteq P(S.f(int), \langle \text{this} \rangle, \langle S.o \rangle)
\end{align*}
\]

After one iteration, the final results are obtained:

\[
\begin{align*}
S.f(int), \#18 &\in P(S.f(int).a) \\
S.f(int), \#18 &\in P(A.f(int), \langle \text{init} \rangle, \langle \text{this} \rangle) \\
S.f(int), \#18 &\in P(C, \langle \text{init} \rangle , \langle \text{this} \rangle) \\
P(A.f(Object->Object), \langle \text{return} \rangle) \\
&\subseteq P(S.f(int), \langle \text{this} \rangle, \langle S.o \rangle)
\end{align*}
\]

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\(^6\) 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 altogether 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: \[ I \equiv \text{new } A \quad | \quad S = [\ldots] \]
\[
\begin{align*}
m.\text{adr}(I) &\in \text{Obj} \\
type(m.\text{adr}(I)) &= A \\
S &= [m.\text{adr}(I), \ldots]
\end{align*}
\]

Assignment: \[ I \equiv \text{astore } r \quad | \quad S = [p_1, \ldots] \]
\[
\begin{align*}
(R\text{egister}(m, r, b + 1), p) &\in \text{Ass} \\
R\text{egister}(m, r, b + 1) &\in \text{Ptr} \\
S &= [\ldots]
\end{align*}
\]

Virtual call (static part): \[ I \equiv \text{invokevirtual } m \quad | \quad S = [p_r \ldots p_1, q, \ldots] \]
\[
\begin{align*}
m.\text{ret}/q &\in \text{Ptr} \\
type(m.\text{ret}/q) &= t \\
S &= [m.\text{ret}/q, \ldots]
\end{align*}
\]

Virtual call (conditional constraints):
\[
\begin{align*}
I \equiv \text{invokevirtual } m \\
sig(m) &\Rightarrow (t_1, \ldots, t_n) \rightarrow t \\
m' &= \text{Lookup}(o, m) \\
S &= [\ldots]
\end{align*}
\]
\[
\begin{align*}
\forall i \in \text{par}(m', i), p_i &\in \text{Ass}, \\
(m', o.i, p_i) &\in \text{Ass}, \\
(type(m', o.i), p_i) &\in \text{Ass} \\
m'.\text{this} &\in \text{Ptr} \\
type(m'.\text{this}) &= \text{ds}(m') \\
m'.\text{ret}/q &\in \text{Ptr} \\
type(m'.\text{ret}/q) &= t \\
o \in \text{Ptr}(q) \Rightarrow \\
&[\ldots]
\end{align*}
\]

Data member access (static part): \[ I \equiv \text{getfield } f \quad | \quad S = [p_i] \\
f' &= \text{LookupField}(f) \\
S &= [p_i] \\
\]
\[
\begin{align*}
p.f' &\in \text{Ptr} \\
type(p.f') &= \text{type}(f') \\
S &= [p, f', \ldots]
\end{align*}
\]

Data member access (conditional constraints): \[ o \in \text{Ptr}(p) \Rightarrow \\
I \equiv \text{getfield } f \\
f' &= \text{LookupField}(f) \\
\]\n\[
\begin{align*}
(p.f', o.f') &\in \text{Ass} \\
S &= [(p.f', o.f')] \\
type(o.f') &= \text{type}(f') \\
o \in \text{Ptr} \Rightarrow \\
\]
\[
\begin{align*}
I \equiv \text{getfield } f \\
f' &= \text{LookupField}(f) \\
\end{align*}
\]
\[
\begin{align*}
(p.f', o.f') &\in \text{Ass} \\
S &= [(p.f', o.f')] \\
type(o.f') &= \text{type}(f') \\
o \in \text{Ptr} \Rightarrow \\
\]
\[
\begin{align*}
I \equiv \text{aastore } f \\
\]
Figure 6: A program fragment, its Bytecode, and the corresponding application of constraint-generating rules.

\[
\begin{align*}
\text{m is unanalysed,} & \quad \text{sig}(m) = (t_1, \ldots, t_n) \rightarrow t \\
\forall_{i=1}^n \text{unanalysed, } m.<\text{par}(m, i)> & \in \text{Ass} \\
\end{align*}
\]

\[
\begin{align*}
\text{m is unanalysed,} & \quad \text{sig}(m) = (t_1, \ldots, t_n) \rightarrow t \\
(m.<\text{ret}>, \text{unanalysed}) & \in \text{Ass} \\
\end{align*}
\]

\[
\begin{align*}
o \in \text{Pl}(\text{unanalysed}/t) \Rightarrow & \quad \text{m is method in t,} \\
& \quad \text{m is visible in unanalysed code,} \\
& \quad \text{sig}(m) = (t_1, \ldots, t_n) \rightarrow t \\
& \quad m' = \text{lookup}(m, o) \\
\forall_{i=1}^n \text{unanalysed, } m.<\text{par}(m, i), \text{unanalysed}> & \in \text{Ass} \\
(m', <\text{this}, o>) & \in \text{Ass} \\
(\text{unanalysed, } m'.<\text{ret}>) & \in \text{Ass} \\
\end{align*}
\]

Figure 7: Constraint generation for unanalysed functions
their parameters; in particular any object given to an un-analysed 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 unanalysed function, an assignment (unanalysed, \( p \)) is added. Pointers which are returned from unanalysed functions could analogously modelled as assignments (\( p, \text{unanalysed} \)). 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, for every type \( t \). The relation between unanalysed and the various unanalysed, is given by

\[
o \in Pt(\text{unanalysed}) \land \text{type}(o) \leq t \Rightarrow o \in Pt(\text{unanalysed})
\]

Instead of inserting the assignment (\( p, \text{unanalysed} \)) whenever the return value of an unanalysed function is assigned to \( p \), we insert (\( p, \text{unanalysed} \)), 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 \( \text{invokevirtual} \) 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 \( \text{classList} \). In order to analyse the call \( \text{c.n.getClass()} \), we first determine \( Pt(c) \). The types of all objects in this set determine which \( \text{classList} \) have to be added to \( Pt(c) \). In the example, \( Pt(c) = \{ \text{classList} \} \). Therefore the call \( \text{c.newInstance()} \) will return a new special object \( \text{dyn : a of type A} \). As a result, the call \( \text{a2.f()} \) 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 traversal 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 explicitly 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:

\[
P_{tsn}(p) = \{ o \in \text{Obj} \mid \text{type}(o) \leq \text{type}(p) \}
\]

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

\[
RP = \frac{\sum_{p \in Pt} |Pt(p)|}{\sum_{p \in Pt} |Pt_{tsn}(p)|}
\]

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: \((p, q) \in Ass \Rightarrow (Pt(q) \subseteq Pt(p)) \iff (p, q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow o \in Pt(p)) \iff (p, q) \in Ass \Rightarrow (o \in Pt(q) \Rightarrow (p, o) \in Ass)\). 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

<table>
<thead>
<tr>
<th>Program</th>
<th>Cl.</th>
<th>Me.</th>
<th>code</th>
<th>calls</th>
<th>static</th>
<th>time</th>
<th>RP</th>
<th>sec</th>
<th>res.</th>
<th>client</th>
<th>client-t.</th>
<th>time</th>
<th>RP</th>
<th>res.</th>
<th>client</th>
<th>client-t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haar</td>
<td>17</td>
<td>230</td>
<td>23k</td>
<td>1011</td>
<td>94.0</td>
<td>51.53</td>
<td>71.7</td>
<td>0/3</td>
<td>5.9</td>
<td>16.2</td>
<td>13.01</td>
<td>59.98</td>
<td>95.0</td>
<td>4.3</td>
<td>16.9</td>
<td>13.73</td>
</tr>
<tr>
<td>JComputer</td>
<td>63</td>
<td>390</td>
<td>39k</td>
<td>2261</td>
<td>94.2</td>
<td>162.31</td>
<td>58.9</td>
<td>14/14</td>
<td>5.1</td>
<td>22.8</td>
<td>55.91</td>
<td>282.21</td>
<td>84.6</td>
<td>4.9</td>
<td>22.9</td>
<td>59.60</td>
</tr>
<tr>
<td>JBInHex</td>
<td>5</td>
<td>54</td>
<td>3024</td>
<td>75</td>
<td>81.3</td>
<td>7.12</td>
<td>77.7</td>
<td>2/2</td>
<td>18.7</td>
<td>33.2</td>
<td>1.11</td>
<td>5.68</td>
<td>96.8</td>
<td>6.7</td>
<td>33.4</td>
<td>1.14</td>
</tr>
<tr>
<td>JLex</td>
<td>26</td>
<td>161</td>
<td>28k</td>
<td>1063</td>
<td>97.7</td>
<td>45.24</td>
<td>67.4</td>
<td>2/7</td>
<td>13/13</td>
<td>19.0</td>
<td>8.96</td>
<td>75.72</td>
<td>94.1</td>
<td>1.9</td>
<td>20.2</td>
<td>8.87</td>
</tr>
<tr>
<td>Jccodes</td>
<td>8</td>
<td>41</td>
<td>2897</td>
<td>158</td>
<td>68.4</td>
<td>19.98</td>
<td>74.7</td>
<td>2/2</td>
<td>21.5</td>
<td>23.5</td>
<td>3.05</td>
<td>7.40</td>
<td>95.7</td>
<td>21.5</td>
<td>23.5</td>
<td>3.07</td>
</tr>
<tr>
<td>NanoXML</td>
<td>3</td>
<td>32</td>
<td>2296</td>
<td>158</td>
<td>96.2</td>
<td>6.64</td>
<td>81.0</td>
<td>2/2</td>
<td>3.8</td>
<td>16.0</td>
<td>1.09</td>
<td>4.89</td>
<td>98.4</td>
<td>3.8</td>
<td>16.0</td>
<td>1.12</td>
</tr>
<tr>
<td>ProxyHammer</td>
<td>12</td>
<td>38</td>
<td>3079</td>
<td>242</td>
<td>88.8</td>
<td>14.76</td>
<td>73.9</td>
<td>0/0</td>
<td>10.7</td>
<td>31.8</td>
<td>2.74</td>
<td>11.02</td>
<td>98.5</td>
<td>10.7</td>
<td>31.8</td>
<td>2.68</td>
</tr>
<tr>
<td>TextScroll</td>
<td>4</td>
<td>92</td>
<td>6644</td>
<td>425</td>
<td>73.2</td>
<td>16.71</td>
<td>68.0</td>
<td>0/0</td>
<td>26.8</td>
<td>6.4</td>
<td>4.35</td>
<td>14.02</td>
<td>95.6</td>
<td>6.4</td>
<td>4.35</td>
<td>4.39</td>
</tr>
<tr>
<td>TumblingDice</td>
<td>34</td>
<td>196</td>
<td>12k</td>
<td>762</td>
<td>90.8</td>
<td>50.12</td>
<td>73.6</td>
<td>2/2</td>
<td>8.8</td>
<td>17.7</td>
<td>13.37</td>
<td>47.10</td>
<td>97.0</td>
<td>8.8</td>
<td>17.7</td>
<td>13.13</td>
</tr>
<tr>
<td>arabeske</td>
<td>21</td>
<td>296</td>
<td>41k</td>
<td>1964</td>
<td>82.9</td>
<td>162.75</td>
<td>76.6</td>
<td>3/5</td>
<td>4.3</td>
<td>16.5</td>
<td>18.85</td>
<td>163.21</td>
<td>94.7</td>
<td>4.3</td>
<td>16.5</td>
<td>19.18</td>
</tr>
<tr>
<td>graph</td>
<td>32</td>
<td>228</td>
<td>16k</td>
<td>1192</td>
<td>96.7</td>
<td>47.98</td>
<td>66.9</td>
<td>1/3</td>
<td>3.2</td>
<td>24.2</td>
<td>14.78</td>
<td>71.48</td>
<td>96.4</td>
<td>3.2</td>
<td>24.2</td>
<td>14.99</td>
</tr>
<tr>
<td>hanoi</td>
<td>45</td>
<td>362</td>
<td>21k</td>
<td>1005</td>
<td>82.5</td>
<td>43.96</td>
<td>72.2</td>
<td>2/1</td>
<td>13.5</td>
<td>20.1</td>
<td>17.37</td>
<td>44.81</td>
<td>95.2</td>
<td>13.0</td>
<td>20.7</td>
<td>14.78</td>
</tr>
<tr>
<td>j502</td>
<td>1</td>
<td>31</td>
<td>8123</td>
<td>78</td>
<td>100.0</td>
<td>3.21</td>
<td>40.3</td>
<td>0/0</td>
<td>0.0</td>
<td>10.7</td>
<td>0.48</td>
<td>1.73</td>
<td>98.0</td>
<td>0.0</td>
<td>10.7</td>
<td>0.49</td>
</tr>
<tr>
<td>jEdit</td>
<td>108</td>
<td>489</td>
<td>34k</td>
<td>2179</td>
<td>81.1</td>
<td>430.12</td>
<td>70.7</td>
<td>23/23</td>
<td>13.9</td>
<td>20.3</td>
<td>117.15</td>
<td>473.89</td>
<td>88.6</td>
<td>13.2</td>
<td>20.3</td>
<td>115.39</td>
</tr>
<tr>
<td>jas</td>
<td>127</td>
<td>435</td>
<td>26k</td>
<td>1042</td>
<td>85.1</td>
<td>611.56</td>
<td>84.5</td>
<td>12/24</td>
<td>3.7</td>
<td>4.9</td>
<td>54.18</td>
<td>404.69</td>
<td>99.6</td>
<td>3.7</td>
<td>4.9</td>
<td>54.10</td>
</tr>
<tr>
<td>java_cup</td>
<td>41</td>
<td>396</td>
<td>32k</td>
<td>2362</td>
<td>93.2</td>
<td>142.71</td>
<td>71.6</td>
<td>5/11</td>
<td>3.4</td>
<td>19.5</td>
<td>35.58</td>
<td>301.58</td>
<td>97.4</td>
<td>3.4</td>
<td>19.5</td>
<td>33.54</td>
</tr>
<tr>
<td>jaxp</td>
<td>110</td>
<td>761</td>
<td>39k</td>
<td>1579</td>
<td>78.3</td>
<td>528.30</td>
<td>84.1</td>
<td>27/93</td>
<td>7.6</td>
<td>11.9</td>
<td>138.74</td>
<td>596.49</td>
<td>96.8</td>
<td>6.2</td>
<td>12.0</td>
<td>134.03</td>
</tr>
<tr>
<td>jflex</td>
<td>52</td>
<td>418</td>
<td>50k</td>
<td>2266</td>
<td>96.9</td>
<td>196.77</td>
<td>81.3</td>
<td>4/11</td>
<td>2.6</td>
<td>17.6</td>
<td>42.24</td>
<td>373.38</td>
<td>95.5</td>
<td>2.5</td>
<td>17.6</td>
<td>38.53</td>
</tr>
<tr>
<td>jpringles</td>
<td>13</td>
<td>71</td>
<td>8045</td>
<td>239</td>
<td>90.8</td>
<td>15.19</td>
<td>68.3</td>
<td>2/2</td>
<td>8.4</td>
<td>15.6</td>
<td>3.17</td>
<td>7.68</td>
<td>97.4</td>
<td>8.4</td>
<td>15.6</td>
<td>2.98</td>
</tr>
<tr>
<td>mars</td>
<td>19</td>
<td>120</td>
<td>5431</td>
<td>371</td>
<td>92.7</td>
<td>39.04</td>
<td>64.4</td>
<td>2/4</td>
<td>2.4</td>
<td>18.1</td>
<td>8.51</td>
<td>28.02</td>
<td>92.4</td>
<td>2.4</td>
<td>18.3</td>
<td>8.42</td>
</tr>
<tr>
<td>ablecc</td>
<td>283</td>
<td>1867</td>
<td>74k</td>
<td>4562</td>
<td>68.2</td>
<td>949.04</td>
<td>62.3</td>
<td>2/72</td>
<td>20.1</td>
<td>4.0</td>
<td>363.71</td>
<td>1069.11</td>
<td>88.5</td>
<td>17.7</td>
<td>4.0</td>
<td>343.45</td>
</tr>
<tr>
<td>yamm</td>
<td>71</td>
<td>264</td>
<td>39k</td>
<td>3279</td>
<td>89.8</td>
<td>1270.15</td>
<td>58.5</td>
<td>5/6</td>
<td>7.1</td>
<td>24.5</td>
<td>93.62</td>
<td>1390.87</td>
<td>78.5</td>
<td>6.2</td>
<td>24.5</td>
<td>91.84</td>
</tr>
</tbody>
</table>

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
destroy 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 aver-
age 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 num-
ber of statically resolved method calls. The columns “res.”
give the percentage of calls which could not be resolved sta-
tically, but where the points-to set is so small that the tar-
get method is unique. Again, the subset-based method is
better. Adding the values in column “static” and in col-
umn “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]. Routev reports very pos-
itive 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 be-
fore 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 typi-
cal for OO programming prevent the approach from being
effective in Java.

Let us finally consider the effect of the different methods
on a specific client analysis, namely the KABA system as de-
scribed 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 ev-
ery 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
Routev, 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 ap-
proximations for unanalysed code and the reflection API.
Routev et al. use Scoot7 as a frontend and the BANE sys-
tem [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]).

7. CONCLUSION AND FUTURE WORK
We presented a comparison of a subset-based and a unifica-
tion-based points-to approach for Java. Our results can be sum-
marized as follows:

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

While in the world of imperative languages such as C, Steen-
sgaard’s method is much faster than Andersen’s, intrapro-
dural Steensgaard combined with fixpoint iteration for dy-
namic binding is slightly slower for Java. The reason is
that the fixpoint iteration leads to a propagation of the im-
precision in Steensgaard’s method, and eventually to slower
convergence.

There is still much room for improvement, both in preci-
sion 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 (csp. [12]). Context-sensitive points-to analysis for
Java can be achieved in a way analogously 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 per-
formance/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 pos-
sible 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 already been done [16]; for Java, it
remains to be seen whether it is possible.

7http://www.sable.mcgill.ca

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


