Efficient Local Type Inference

3rd Year Project Report

Benjamin Bellamy, Magdalen College

Trinity Term, 2008

Abstract

I consider the problem of local type inference, where the types of local variables are inferred within a method with otherwise complete static type information. This is an important problem for tools which manipulate languages, such as Java bytecode, where local type information does not exist. Another application of local type inference would enable the design of programming languages where the types of local variables need not be declared by the programmer. Even when programmer-declared types for a local variable are available, it is possible that these may not be as tight as possible. Some analyses on programs, such as generation of a call graph, give more useful results when local variables are typed as tightly as possible. Finally local type inference can be seen as a sub-problem in global type infereuce for object-oriented languages, where not even method signatures are available.

I construct a new algorithm, built upward from a definition of optimal typing validity. I begin by examining the Java bytecode verifier, which is perhaps the 'most executed' example of local type inference algorithm. I consider how the bytecode verifier solves a similar problem to local type inference, but is in some aspects quite different. I use some of these ideas iu the development of my algorithm.

I derive a 'core' algorithm for local type inference in a language that obeys certain requirements, and prove this correct. Then I go on to consider how the algorithm can be generalized further, relaxing certain requirements on the target language. This yields a final algorithm, general enough for local type inference that is a specific target language: Jimple, which is 'somewhere between' Java bytecode and Java source.

Through extensive experiments on over 295K Jimple methods, generated by a range of different compilers. I show that my algorithm is typically around 4 to 5 times faster than algorithms currently in use. I show that although my algorithm has exponential worst-case complexity, it exhibits linear complexity in common cases. Other algorithms offer better worst-case complexity but are usually slower in practice.

A paper on this project has been accepted at the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2008). A draft copy of that paper is attached. The role of the coauthors has been to guide this research and help with the presentation of the paper, for instance with a survey of the literature. All the novel ideas for the algorithm, its formalization and its evaluation are my own.

This project text is shorter than 10,000 words; but all figures, equations, algorithms and proofs should be considered 'extra material'. These are included in line with the text for ease of reference.

Contents

1 Introduction

Type inference is the process of automatically inferring types for a computer program. This work specifically deals with *local type inference* in object-oriented languages, where least static types are inferred for local variables within a method, given all other typings such as method parameter types, return types and public field types.

Some details of this work are specific to the Jimple programming language [7], where there is a particular requirement for local type inference, though most of the algorithms presented could be easily adapted to most modern object-oriented languages. Jimple was introduced with the Soot framework [7] as a statically typed, stack-less *intermediate language* between Java bytecode and Java source. Most statements in Jimple take at most three variables, so Jimple is sometimes described as a 'three-address representation'. The intention of Soot's authors was to design a language that made it easier to analyze and transform Java bytecode. Java bytecode can be translated to Jimple more easily than it can be fully decompiled to Java source, though Jimple also provides a useful intermediate stage in the full decompilation of Java. One particular difference between Java bytecode and Jimple is that bytecode uses an operand stack as well as a local variable array; Jimple only supports local variables. In bytecode to Jimple conversion, elements on the operand stack are replaced by local variables, and then all local variables are split as much as possible by flow analysis without inserting 'copy' statements. This splitting does not, in general, result in Single Static Assignment (SSA) form because assignments to the same variable are allowed in parallel branches of flow-control statements.

Finally, after the splitting step, a static type is assigned to each of the local variables. There is no type information for the operand stack or local variable array in Java bytecode, so these static types in Jimple must be inferred from the type information that is available. The original objective for this project was to develop a method for this final local type inference stage of the bytecode to Jimple translation. In creating Soot, Gagnon *et al.* [2] designed an algorithm for local type inference in Jimple. Their algorithm exhibits polynomial worst-case complexity. I present an algorithm that has theoretical exponential worst-case complexity, but experiments show that it is typically linear and significantly faster in most cases tested. Also my algorithm is proved to find a tightest valid typing in all cases, whereas experiments reveal that Gagnon's algorithm occasionally does not.

But most of the work I present is not limited to this specific application to Jimple. The algorithms are general enough to be used with any typical object-oriented programming language. For example, the newest versions of Microsoft's C# and Visual Basic languages include local type inference, but only where a variable is declared and defined on one line. Using a more complete local type inference algorithm, such as that presented here, declared static types would not be required at all, yet the compile-time verification and run-time performance benefits of strong typing would be maintained.

Another application of local type inference is in program analysis. Even if programmer-declared types for local variables are available, these may be weaker than strictly necessary. This has an adverse affect on some kinds of analysis, such as constructing a call graph using Class Hierarchy Analysis (CHA). Here we need to determine which concrete methods can be invoked from each virtual function call $b.f(\cdots)$. The ideal result is the set containing each method f in all classes ever referenced by b at the method call. The challenge is finding out exactly which classes these are. In CHA the type of b is used, and we know that b can only contain subtypes of the type of b, so we find that the method called could be the function f in any of these subtypes. Clearly if b has a tighter type then there will be fewer classes that b could reference, so a smaller and more useful call graph is found. So local type inference could be used, even where programmer-declared types are available, to find the tightest typing for each variable and thus the smallest call graph by CHA.

Finally, the algorithms I present could form a subroutine of a global type inference procedure, where the types in method signatures are also inferred. There is a large literature on the subject of global type inference; the greatest advances were made by Palsberg and Schwartzbach [6]. In my conclusions I briefly comment on possible future work, where my algorithm could be extended to solve the global type inference problem.

1.1 Overview

I begin in Section 2 by examining the Java Bytecode Verifier, and consider the similarities and differences between the problems of bytecode verification and of local type inference. I then formally describe the problem in relation to a 'simplified language', which excludes some typical features of object-oriented languages such as arrays and multiple inheritance. I first use intuition to develop an algorithm for this simplified language, and present a proof that this algorithm solves my formal description of the problem.

Next in Section 3 I remove a significant restriction on the simplified language by allowing multiple inheritance. I modify the formal description of the problem, and again use intuition to give a more general type inference algorithm. Finally I present a new proof, which is somewhat more complex.

Section 4 offers more generalizations to the algorithm, relaxing further the requirements of the language. This introduces support for Java-style arrays, modifications to methods that are not immediately typable, and expressions that have more than one (least) type in the hierarchy. The general algorithm given by this section is capable of local type inference in the Jimple language and I implement this for evaluation. I use this implementation to conduct a thorough set of experiments in Section 5.

Section 6 summarizes some of the most closely related work and compares them to my algorithm. Finally I conclude in Section 7, in which I briefly comment on possible future work. I also present my personal report, discussing the project, the challenges involved, and what I have learned in the process.

2 Algorithm Design and Development

2.1 The Java Bytecode Verifier

I begin by considering the Java bytecode verifier, which is ~~is~~ part of any Java virtual machine following Sun's specification [5]. The intended purpose of bytecode verification is to check that the bytecode is structurally well formed and well behaved at runtime. Part of this verification ensures that no local variable or operand stack element could ever be used when it contains a value of an incompatible type. Remember that the `null` type is allowed to be 'used' *at compile-time* wherever an object reference or array is expected.

Briefly, this interesting part of the bytecode verifier works by developing a typing for the local variable array and operand stack at each instruction in a method. If at any point the typing is not valid, based on the conversions allowed by the specification, then the verifier fails. A pseudo-code algorithm for the bytecode verifier is given as Algorithm 1. In this pseudo-code, and through the rest of this discussion, I ignore the Java bytecode operand stack. The actual bytecode verifier does perform similar verification on the operand stack, but this work only considers local type inference in languages without such a stack. Indeed, the bytecode to Jimple translation inserts local variables to replace all operand stack locations.

Initially the verifier sets the typing at the first instruction of the method to contain initial types for method parameter variables. All other variables map to a special 'top' type $\top$ indicating that the variable is unusable. The typings of all other instructions are set to a special 'bottom' typing $\sigma_\perp$, indicating that the instruction has not yet been visited. The verifier then adds the first instruction in the method to a worklist of instructions that need to be examined. While the worklist is not empty the algorithm continues in a loop. It removes an instruction from the worklist, checks that the typing is valid, and verification fails if not. It then creates a new typing, modeling how the instruction's typing will change after execution of the instruction. This new typing is then merged with the typings of any possible subsequent instructions considering control flow. Two types are merged by taking the least-common-ancestor of the two types, which is the least type that can hold all values of both types.

So in the bytecode verifier, at each statement (instruction) we have a typing for the local variables, though this typing is not generally valid at any other statements. For example, at one instruction local variable `x` might contain a value of type `Vector`, and then at another instruction it may contain a `String`. This is because we are dealing with Java bytecode and not Java source, so all variables do not need a single static type.

So it turns out that a correct local type inference algorithm will look a little different from the algorithm of the bytecode verifier. Primarily we will probably not consider control flow. A valid typing must be valid for all statements, not just the statements that might 'see' that typing.

Algorithm 1 Bytecode verification

```
1: for all instructions $i$ in method do
2:    $typing(i) \Leftarrow \sigma_\perp$
3: end for
4: $i \Leftarrow$ { first instruction in method }
5: $worklist \Leftarrow \{i\}$
6: $typing(i) \Leftarrow$ typing containing correct method parameter types, all other local variables map to $\top$
7: while $worklist \neq \emptyset$ do
8:    $i : worklist \Leftarrow worklist$
9:    if $i$ is not well typed under $typing(i)$ then
10:      verification fails
11:   end if
12:   $\sigma \Leftarrow$ effect of $i$ applied to $typing(i)$
13:   for all instructions $i_{next}$ that can follow $i$ in control flow do
14:      if $typing(i_{next}) = \sigma_\perp$ then
15:         $typing(i_{next}) \Leftarrow \sigma$
16:         $worklist \Leftarrow i_{next} : worklist$
17:      else
18:         for all local variables $v$ do
19:            if $typing(i_{next})(v)$ and $\sigma(v)$ are both reference types then
20:               $\sigma'(v) \Leftarrow$ least common superclass of $typing(i_{next})(v)$ and $\sigma(v)$
21:            else if $typing(i_{next})(v) \neq \sigma(v)$ then
22:               $\sigma'(v) \Leftarrow \top$
23:            end if
24:         end for
25:         if $\sigma' \neq \sigma$ then
26:            $typing(i_{next}) \Leftarrow \sigma'$
27:            $worklist \Leftarrow i_{next} : worklist$
28:         end if
29:      end if
30:   end for
31: end while
32: Verification succeeds
```

2.2 Local Type Inference in a Simplified Language

We can begin to formalize the problem. Let T be a set of all types visible to the method, with a subtype relation $\leq$ between them. The set T must have an infimum and supremum. In this report I sometimes refer to the infimum and supremum of all types in T as $\perp$ (bottom, or untyped) and $\top$ (top, or failure) respectively. $\perp$ is a subtype of every type and $\top$ is a supertype of every type. It is usually possible to extend the type hierarchy with new imaginary types $\perp$ and $\top$ if real types do exist. Sometimes properties of the target language will ensure that these types are never provided by the type inference algorithms, and I mention these cases when describing each algorithm later in this report. But, to give an example, the fact that every variable is assigned at least once in Jimple means that no variable is ever typed at $\perp$. Also (except for small integer types, which I consider mnch later) Jimple guarantees that no variable is ever assigned incompatible types, so no variable is ever typed as $\top$.

Next, let V denote the set of local variables in the method. We define a *typing* as a mapping from local variables to types $V \mapsto T$.

First of all I consider methods in a *simplified language* that satisfy some requirements. I formally define these requirements later in this section, but the list below might be a useful summary. It turns out that general Jimple methods satisfy none of these, so later in this report I generalize the algorithm for them, removing each requirement in turn.

- The $\leq$ relation on T forms a lattice, so any pair of types has a single Greatest-Lower-Bound (GLB) and Least-Upper-Bound (LUB) (remember T inclndes $\perp$ and $\top$.) LUB is also known as *least-common-ancestor* or *join*, and can be represented by the $\vee$ operator. For the algorithms I present the least-common-ancestor function lca : $T^2 \mapsto T$ must be well defined. The requirement that types form a lattice is actually a slightly weaker requirement than a total absence of multiple inheritance, though there is certainly no single-valued *lca* function for the Java (and Jimple) hierarchy. [This requirement is removed in Section 3]
- A 'valid' typing does exist for the method. [This requirement is removed in Section 4.1]
- Only single local variables or field references, and in particular *not* arrays, can appear on the left-hand-side of assignment statements. [Java-style arrays are supported in Section 4.2]
- A single-valued monotonic function eval : $\Sigma \times E \mapsto T$ is well defined. The intuitive meaning of eval(σ, e) is to infer the type of expression e under typing σ. [A multi-valued eval function is supported in Section 4.3.3]

I now describe these requirements formally. We require that least-common-ancestor function lca must satisfy the following equivalence for all $x, y, z \in T$:

$$x \leq z \wedge y \leq z \equiv \mathrm{lca}(x, y) \leq z \tag{1}$$

By taking $z = \mathrm{lca}(x, y)$ we observe that for all $x, y \in T$

$$x \leq \mathrm{lca}(x, y) \wedge y \leq \mathrm{lca}(x, y) \tag{2}$$

We can verify that the lca function is associative and commutative. In our simplified language, local variables can appear in only one of two contexts:

- *assignments* of the form $v := e$, the set of which we name A,
- and *uses*, which can take several syntactic forms, but always convert a variable v to some type t. We model this use by the pair (v, t), and we denote the set of all uses as U.

We can define a typing as a mapping from V to T, and a partial order $\leq$ on typings as

$$\sigma_1 \leq \sigma_2 \equiv \forall v \in V, \sigma_1(v) \leq \sigma_2(v) \tag{3}$$

We require that the language exhibits the following common definitions of typing validity:

- an assignment $v := e$ is *valid* under a typing σ if and only if eval$(\sigma, e) \leq \sigma(v)$,

- a use (v, t) is valid under a typing σ if and only if $\sigma(v) \leq t$,
- a method (A, U) is valid under a typing σ if and only if all assignments in A and all uses in U are valid under σ.

Here I introduce the eval : $\Sigma \times E \mapsto T$ function to formalize validity. I shall require that the eval function ~~is~~ be monotonic, so

$$\sigma_1 \leq \sigma_2 \implies \text{eval}(\sigma_1, e) \leq \text{eval}(\sigma_2, e) \tag{4}$$

It is the definitions of typing validity, along with language-dependent lca and eval functions, which provide the 'link' between type my algorithm and the target programming language. I make no other assumptions about the language until I extend the algorithm to support additional language features in later sections.

It is clear from the above definitions that the problem of local type inference for a method can be defined as finding the least typing under which the method is valid. For convenience I write that a typing σ is *assignment-valid* for a method if and only if every assignment is valid under σ, formally

$$\forall (v := e) \in A, \text{eval}(\sigma, e) \leq \sigma(v) \tag{5}$$

Likewise a typing σ is *use-valid* for a method if and only if every use is valid under σ, formally

$$\forall (v, t) \in U, \sigma(v) \leq t$$

It is obvious that a clear that maps all local variables to $\top$ is assignment-valid, so we know that a least assignment-valid typing must exist. Now notice how the definition of assignment-validity (5) is somewhat related to the definition of the *lca* function (1). I leave the proof that (under the simplified language) the least assignment-valid typing is unique until Section 2.3.1, but it may seem intuitive to the reader. For now I continue with an informal algorithm derivation, assuming that a unique least assignment-valid typing exists.

With this assumption I can make an important observation: **if the least assignment-valid typing is not use-valid, then there are no valid typings.** We can see this easily by considering the least assignment-valid typing σ. If σ is not use-valid then, for some use (v, t), $\sigma(v) \not\leq t$. Now all typings *not* greater than σ are not assignment-valid because σ is the least. Finally consider any typing σ' greater than σ. By definition, $t \not\leq \sigma(v)$ so $t \not\leq \sigma'(v)$, so σ' is not use-valid either.

Armed with this observation I realize that is it sufficient to find the least assignment-valid typing and then check that it is use-valid. This gives the least valid typing.

2.3 Finding the Least Assignment-Valid Typing

In this section I present several 'tries' at an algorithm. After each attempt I explain what is wrong, giving an example, and fix it to give another algorithm. This roughly follows my thought process when I was designing the algorithm. For the final algorithm, a formal proof of correctness follows in Section 2.3.1.

Suppose we maintain a typing σ and iterate through the assignment statements individually, updating σ as we progress. A first attempt at an algorithm might look like the algorithm in Figure 1.

```
1: for all assignments $v := e$ do
2:     $\sigma \Leftarrow \sigma[v \mapsto \text{eval}(\sigma, e)]$
3: end for
```

```
<untyped> x;
x = new Integer(5);
x = new String("Some String");
```

Figure 1: Algorithm Attempt 1 and Counterexample

Of course this is not sufficient. There may be several assignments with the same local variable v on the left hand side, and we must ensure that v is given the least type that makes valid all such assignments. In this case the least such type is `Object`. This is given by the least-common-ancestor function lca : $T^2 \mapsto T$, which we know is well-defined in this simplified language. Figure 2 shows a second attempt. This time we start with all local variables typed as $\bot$. I don't attempt to prove this algorithm rigorously at this stage, but given that $\forall x \in T, lca(\bot, x) = x$, and remembering the associativity and commutativity of lca, one might be persuaded that we are on the right track. Also, as long as each variable is at some point assigned a value of some type other than $\bot$, as is the case in Jimple, then the final typing will never map any locals to $\bot$.

```
for all local variables $v$ do
    $v \Leftarrow \bot$
end for
for all assignments $v := e$ do
    $\sigma \Leftarrow \sigma[v \mapsto \text{lca}(\sigma(v), \text{eval}(\sigma, e))]$
end for
```

```
1   <untyped> x, y;
2   x = new Integer(5);
3   y = x;
4   x = new String("Some String");
```

Figure 2: Algorithm Attempt 2 and Counterexample

Indeed Figure 2 is closer but we're not there yet. Since σ is changing, in fact monotonically increasing, then the value of eval(σ, e) for any expression e might also be changing. This can happen when a local variable appears in the expression on the *right-hand-side* of an assignment. In the example we would type `x` as `Integer` on line 2, `y` as `Integer` on line 3, then `x` as `Object` on line 4. So at the end line 3 is no longer valid. What the algorithm should do in this case is return to reconsider line 3 whenever the type of `x` changes.

ApplyAssignmentConstraints (Algorithm 2) is a complete local type inference algorithm for the simplified language. The algorithm takes a typing parameter σ, and returns a singleton set containing the least assignment-valid typing that is greater than σ. The reason for returning a singleton is to maintain compatibility with more general versions of the algorithm that will return sets of typings instead. For the current purpose we will always take $\sigma = \sigma_\bot$, the infimum of all typings, where every variable maps to $\bot$. This parameter is used in later applications of the algorithm.

The algorithm maintains a set *worklist* of assignments that we still need to consider. Initially *worklist* is set to all assignments in the method, and whenever a the type for a local variable v changes, we add all assignments in *depends*(v) to *worklist*. *depends* : $V \mapsto 2^A$ maps each local variable v to (a superset of) the set of assignments $v' := e$, where eval(σ, e) depends on $\sigma(v)$.

Algorithm 2 ApplyAssignmentConstraints(σ) Version 1
Local type inference in the simplified language

```
for all local variables $v$ do
    $v \Leftarrow \bot$
end for
$worklist \Leftarrow$ all assignments
while $worklist \neq \emptyset$ do
    $(v := e) : worklist \Leftarrow worklist$
    $t \Leftarrow \text{lca}(\sigma(v), \text{eval}(\sigma, e))$
    if $t \neq \sigma(v)$ then
        $\sigma(v) \Leftarrow t$
        $worklist \Leftarrow worklist + \text{depends}(v)$
    end if
end while
return $\{\sigma\}$
```

The reader may wonder whether this algorithm terminates. Again this is proved in Section 2.3.1, and the proof depends on the monotonicity of the eval function (4). It is worth noting that the efficiency of this algorithm can be optimized for a target language by controlling the order in which elements are selected from *worklist*. For example, in Jimple it is preferable to use a structure like a priority queue for *worklist*, where elements are ordered by the order they appear in the method body. This is because it is more usual for local variable assignments to precede uses.

2.3.1 Proof

In this simplified language the type hierarchy $(T, \leq)$ forms a lattice: a partially ordered set where every nonempty subset has a single lowest-upper-bound (also called a least-common-ancestor) and a single greatest-lower-bound. The usual notation is to denote the lowest-upper-bound, or join, of a subset X by $\vee X$.

We can define a function

$$f(\sigma)(v) = \bigvee\{\text{eval}(\sigma, e) | (v := e) \in A\} \tag{6}$$

Since the eval(σ, e) function is monotonic in σ (4), f is also monotonic. It can also be shown that any assignment-valid typing is a prefix point of f:

$$\begin{array}{ll} & f(\sigma) \leq \sigma \\ \equiv & \{ \text{ definition of } \leq \text{ (3) } \} \\ & \forall v, f(\sigma)(v) \leq \sigma(v) \\ \equiv & \{ \text{ definition of f (6) } \} \\ & \forall v, \bigvee\{\text{eval}(\sigma, e) | (v := e) \in A\} \leq \sigma(v) \\ \equiv & \{ \text{ definition of LUB } \} \\ & \forall v, \forall (v := e) \in A, \text{eval}(\sigma, e) \leq \sigma(v) \\ \equiv & \\ & \forall (v := e) \in A, \text{eval}(\sigma, e) \leq \sigma(v) \end{array}$$

So any assignment-valid typing is a fixed point of f, and any fixed point is an assignment-valid typing. The idea is to find the least such fixed point, which is the least assignment-valid typing. Such a least fixed point exists because of the Knaster-Tarski theorem [3], and we will find it by repeated iteration of f on the infimum $\sigma_\perp$ of all typings.

$$f(f(\cdots f(\sigma_\perp) \cdots))$$

The similarities are clear between this theory and the implementation in ApplyAssignmentConstraints$(\sigma_\perp)$ (Algorithm 2.) The key difference is the use of a worklist in the implementation, which is a simple optimization to avoid evaluating eval and lca more than necessary.

3 Multiple Inheritance

Multiple inheritance is a feature of the type hierarchy in most object-oriented languages. For example, consider the Java (and Jimple) type hierarchy shown in Figure 3. Even though in Java multiple superclasses are not allowed, any class can implement any number of interfaces. In this example types C and D are subtypes of both interfaces IA and IB.

```
1   void multInhrA() {
2           <untyped> x;
3           x = new C();
4           x = new D();
5   }
```

```
1   void multInhrB() {
2           <untyped> x;
3           x = new C();
4           x = new D();
5           expectsAnIA(x);
6   }
```

Now examine the ~~the~~ Jimple I given in method multInhrA above. How should we type variable x? Clearly IA and IB are preferable to Object since they are tighter, but we have no reason to choose either. Both of these are least assignment-valid typings. Now examine method multInhrB, where I add a use (x, IA) in line 5. Now only one of these least assignment-valid typings is also use-valid, and this is clearly the one to choose.

Put another way, the difficulty brought by multiple inheritance is that there is no longer a single-valued least-common-ancestor function between pairs of types so the partial order of typings does not

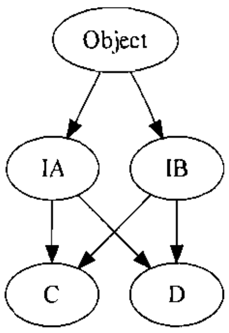


Figure 3: A Type Hierarchy with Multiple Inheritance

form a lattice. Indeed, we can try choosing any of the five types as the value of lca(C, D) and it is easy to check that the required property (1) of the lca function would not hold.

I take the obvious approach to overcoming this difficulty by generalizing the algorithm to support a multi-valued lca function, so lca : $T^2 \mapsto 2^T$. The intuition is that lca(x, y) returns a set of least common supertypes, and a type t is a supertype of both x and y if and only if it is also a supertype of x. The formal requirement of a valid lca function becomes

$$x \leq z \wedge y \leq z \equiv \exists t \in \text{lca}(x, y), t \leq z \tag{7}$$

In the second version of APPLYASSIGNMENTCONSTRAINTS (Algorithm 3) I maintain a set of typings instead of a single typiug. Each iteration involves an assignment and a single typing from the set. We replace that typing with one or more new typings, each accounting for one of the least-common-supertypes.

Notice that the algorithm will return Σ containing all least assignment-valid typings. But as illustrated by the multInhrB example earlier in this section these are not all guaranteed to be use-valid. However we assume that a valid typing does exist, so we select any such typing from Σ to use, as all are least and have equal merit.

3.0.2 Proof

With multiple inheritance in the picture, types no longer generally form a lattice. This is because for some sets of types, as exemplified by Figure 3, there is no single least-upper-bound. Instead we introduce a *step* predicate on pairs typings. Intuitively σ, σ' is in *step* if and only if σ' is still a potential assignment-valid typing when we consider that the right-hand-sides of all assignments will be typed under a typing at least as great as σ. *step* is defined formally as

$$step(\sigma, \sigma') = \forall (v := e) \in A, \text{eval}(\sigma, e) \leq \sigma'(v) \tag{8}$$

We notice that $step(\sigma, \sigma)$ if and only if σ is assignment-valid. One other important property of the *step* predicate is monotonicity in the second argument:

$$step(\sigma, \sigma') \wedge \sigma' \leq \sigma'' \implies step(\sigma, \sigma'') \tag{9}$$

This proof makes use of upward-closed sets of typings, which are defined as sets Σ where

$$\forall \sigma \in \Sigma, \sigma \leq \sigma' \implies \sigma' \in \Sigma \tag{10}$$

The idea is that we use an upward-closed set to represent all potential assignment-valid typings for the method, iteratively removing known invalid typings (from an initial set with all possible typings) as

Algorithm 3 ApplyAssignmentConstraints(σ) Version 2
Local type inference supporting multiple inheritance

```
1: for all local variables $v$ do
2:   $v \Leftarrow \bot$
3: end for
4: $\Sigma \Leftarrow \{\sigma\}$
5: $worklist(\sigma) \Leftarrow$ all assignments
6: while $\exists\sigma \in \Sigma, worklist(\sigma) \neq \emptyset$ do
7:   Pick $\sigma \in \Sigma, worklist(\sigma) \neq \emptyset$
8:   $\Sigma \Leftarrow \Sigma \setminus \{\sigma\}$
9:   $(v := e) : worklist \Leftarrow worklist(\sigma)$
10:  $t' \Leftarrow \text{eval}(\sigma, e)$
11:  for all $t$ in $\text{lca}(\sigma(v), t')$ do
12:    if $t = \sigma(v)$ then
13:      $\Sigma \Leftarrow \Sigma \cup \{\sigma\}$
14:    else
15:      $\sigma' \Leftarrow \sigma[v \mapsto t]$
16:      $worklist(\sigma') \Leftarrow worklist(\sigma) +\!\!+ \text{depends}(v)$
17:      $\Sigma \Leftarrow \Sigma \cup \{\sigma'\}$
18:    end if
19:  end for
20: end while
21: minimize $\Sigma$
22: return $\Sigma$
```

the algorithm progresses. Intuitively we want to find the largest upward-closed set of assignment-valid typings. So we define a partial order on upward-closed sets, and we want to find the least set of typings under this partial order.

$$\Sigma \leq \Sigma' \equiv \Sigma' \subseteq \Sigma \qquad (11)$$

An important property of upward-closed sets is that the entire set can be represented by its minimal elements, which we denote $\text{mnl}(\Sigma)$.

$$\sigma' \in \text{mnl}(\Sigma) \text{ iff } \forall\sigma, \sigma \in \Sigma \wedge \sigma \leq \sigma' \iff \sigma' = \sigma \qquad (12)$$

A property of mnl on upward-closed sets allows us to test for the set inclusion $\Sigma \subseteq \Sigma'$ of two upward-closed sets, when we only know $\text{mnl}(\Sigma)$ and Σ':

$$\Sigma \subseteq \Sigma' \equiv \text{mnl}(\Sigma) \subseteq \Sigma' \qquad (13)$$

The $\Rightarrow$ direction is trivial since $\text{mnl}(\Sigma) \subseteq \Sigma$. The $\Leftarrow$ direction follows from the definition of upward-closed sets (10). Every element σ' that is greater than some element σ of $\text{mnl}(\Sigma)$ must belong to Σ, but since also $\sigma \in \Sigma'$ then $\sigma' \in \Sigma'$.

We are now ready to define a function F, which is a generalization of the f function from Section 2.3.1.

$$F(\Sigma) = \{\sigma' | \exists\sigma \in \Sigma, step(\sigma, \sigma') \wedge \sigma \leq \sigma'\} \qquad (14)$$

By the monotonicity of the *step* predicate we can see that $F(\Sigma)$ is always upward closed. Also all elements of $F(\Sigma)$ are greater than or equal to some element in Σ, so $F(\Sigma) \subseteq \Sigma$ and thus $\Sigma \leq F(\Sigma)$. We finally need to show that any set of assignment-valid typings is a prefix point of F:

$$\begin{aligned}
& F(\Sigma) \leq \Sigma \\
\equiv\ & \{ \text{ definition of } \leq \text{ (11) } \} \\
& \Sigma \subseteq F(\Sigma) \\
\equiv\ & \{ \text{ inclusion of upward-closed sets (13) } \} \\
& \mathrm{mnl}(\Sigma) \subseteq F(\Sigma) \\
\equiv\ & \{ \text{ definition of } F \text{ (14) } \} \\
& \forall \sigma' \in \mathrm{mnl}(\Sigma), \exists \sigma \in \Sigma, \mathit{step}(\sigma, \sigma') \wedge \sigma \leq \sigma' \\
\equiv\ & \{ \text{ definition of mnl (12) } \} \\
& \forall \sigma' \in \mathrm{mnl}(\Sigma), \mathit{step}(\sigma', \sigma')
\end{aligned}$$

So any fixed point of F is an upward-closed set of assignment-valid typings, the least fixed point Σ of F is the set of all assignment-valid typings, and $\mathrm{mnl}(\Sigma)$ is the set of least assignment-valid typings, which is what Algorithm 3 gives.

The starting point for the least fixed point calculation is the least (in $\leq$) upward-closed set of typings. This set may be denoted $\Sigma_\perp$ but is in fact the set containing every possible typing. In the same way as Section 2.3.1 we can find the least fixed point by iteratively evaluating F:

$$F(F(\cdots F(\Sigma_\perp)\cdots))$$

However a naive implementation of this iteration would be horrendously inefficient! Consider having to maintain sets of typings as large as every possible typing! We can perform an extremely rough calculation: the number of types in the Java `rt.jar` file is greater than but in the order of 10000. A typical method with 10 local variables has 10^{10000} possible typings, so this would be the cardinality of $\Sigma_\perp$, and with conventional computing there is no way we could contemplate naive storage of the upward-closed sets of typings.

Examining Algorithm 3 shows that we actually maintain a set of typings Σ' somewhere between $\mathrm{mnl}(\Sigma)$ and Σ, so $\mathrm{mnl}(\Sigma) \subseteq \Sigma' \subseteq \Sigma$. It is sufficient to maintain the set of minimal typings at each step, but experiments show that ensuring we *only* maintain minimal typings is expensive. We minimize once at the end.

4 Jimple-Specific Considerations

ApplyAssignmentConstraints provides the basis for local type inference in most object-oriented languages. But different languages have particular quirks, which may not completely satisfy the requirements of the simplified language used so far. The reader will soon realize that some of the problems presented by these 'quirks' are somewhat difficult to solve in an efficient manner! My initial motivation for this project was to improve the performance of local type inference *in Jimple*, and the generalizations presented in this section are combined at the end to give a complete local type inference algorithm for Jimple.

4.1 Untypable Methods

In languages such as Jimple we are not guaranteed that a valid typing does exist. The reason for this is due to the principle that it must be possible to translate all valid bytecode to Jimple. Ignoring the small integer types, which I consider later, the bytecode to Jimple translator does split variables enough so that an assignment-valid typing (not including $\top$) does always exist, but it is not always able to guarantee a use-valid typing.

Gagnon *et al.* considered this at length and offered some examples of untypable Jimple methods. These are shown in Figure 4. The `untypableA` method will successfully pass the bytecode verifier with `x` dynamically typed as `CA` on line 6, `CB` on line 8 and `Object` on line 9. However there is no static type for `x` that would be valid for all statements. The `untypeableB` method is another example which is untypable, this time due to multiple inheritance. By inspection we can see that the program is well

```
class CA extends Object { void f() { } }
class CB extends Object { void g() { } }
void untypableA() {
        <untyped> x;
        if ( ... )
                { x = new CA(); x.f(); }
        else
                { x = new CB(); x.g(); }
        x.toString();
}
```

```
interface IA { void f(); }
interface IB { void g(); }
interface IC extends IA, IB { }
interface ID extends IA, IB { }
class CC implements IC { void f() { } void g() { } }
class CD implements ID { void f() { } void g() { } }
class InterfaceDemo {
        IC getC() { return new CC(); }
        ID getD() { return new CD(); }
        void untypableB()
    {
                <untyped> x;
                if ( ... ) x = getC(); else x = getD();
                x.f(); x.g();
        }
}
```

Figure 4: Examples of untypable Jimple methods, due to Gagnon *et al.* [2]

behaved, and the bytecode verifier passes because it leaves verification of the Java `invokeinterface` instruction until runtime. But again there is no static typing that will be valid for all statements.

In addition to analyzing the problem, Gagnon *et al.* also presented extensions to their algorithm, which apply semantics-preserving program transformations to 'fix' untypable methods where required, such that a valid typing is guaranteed to exist. I elected to use the same approach. Gagnon gave two stages of transformations: stage A is only ~~be~~ applied if the original method is untypable, and stage B is only applied if the method remains untypable after stage A. After applying stage B a valid typing is guaranteed to exist for every method.

The stage A transformation fixes cases like `untypableA`. This is similar to the method for splitting variables in control flow branches while obtaining Single Static Assignment (SSA) form. But we are allowed multiple assignments, so we do not need to worry about using Φ fuuctions as in SSA form. Wherever an object is instantiated within a control-flow branch we introduce a new variable for the new object, and also immediately 'copy' this reference to the original variable. Now wherever the code contains a use of the original variable, but in the scope of the same control-flow branch, we replace the original variable with the new variable. This allows the new variable to be typed with a more specific type than the old variable. As shown in my experiments (Section 5) Stage A does not fix all untypable methods, an example being `untypeableB`.

The stage B transformation simply inserts casts wherever a use is not valid under a least assignment-valid typing. These casts are guaranteed to succeed at runtime for Jimple (from verifiable bytecode), so we are not altering the program semantics. This stage will always produce a valid typing in Jimple since, as we have already seen, a least assignment-valid typing (not using any $\bot$ or $\top$ types) always exists.

JimpleLocalTypeInference (Algorithm 4) shows how these transformations can be arranged with ApplyAssignmentConstraints. We first try running ApplyAssignmentConstraints($\sigma_\bot$) on the original method, and if no least assignment-valid typing is also use-valid (i.e. it requires at least one cast to make it so) then we apply the stage A transformation. We then re-run JimpleLocalTypeInference($\sigma_\bot$) and select a typing that now requires fewest casts. If this typing is still not use-valid then we apply the stage B transformation by inserting casts wherever they are required. This is guaranteed to give a valid typing.

Algorithm 4 JIMPLELOCALTYPEINFERENCE Version 1
Local type inference including transformations to guarantee a solution

```
 1: $\Sigma \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma_\bot$)
 2: $minCasts \Leftarrow \min\{$COUNTCASTSREQUIRED$(\sigma) | \sigma \in \Sigma\}$
 3: if $minCasts > 0$ then
 4:     Apply stage A transformation
 5:     $\Sigma \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma_\bot$)
 6:     $minCasts \Leftarrow \min\{$COUNTCASTSREQUIRED$(\sigma) | \sigma \in \Sigma\}$
 7: end if
 8: $\sigma \Leftarrow$ any element of $\Sigma$ where $countCasts(\sigma) = minCasts$
 9: Insert casts to make $\sigma$) use valid
10: return $\sigma$
```

4.2 Arrays

So far our language has only allowed assignments of the form $v := e$. Jimple also includes array assignments of the form $v[i] := e$. Intuitively one may be tempted to think we can safely ignore these. After all, every variable that is used as the base of an array reference must contain a 'suitable array'? Not necessarily, as demonstrated in the following snippet:

```
<untyped> x;
x = new String[1];
x[0] = new Object();
```

Clearly there is a problem here, but the bytecode is verifiable because arrays in Java are covariant ($t_1 \leq t_2 \equiv t_1[] \leq t_2[]$.) In fact the Java VM is responsible for keeping track of array types, and would throw an `ArrayStoreException` on line 3. If we *do not* consider line 3 then we would choose to type `x` as `String[]`. We would then need to introduce a cast in statement 3, which would fail at runtime with a `ClassCastException`. If we *do* consider line 3 then we would choose to type `x` as `Object[]`, no casts would be required, and line 3 would fail at runtime with an `ArrayStoreException`. Since we must not change program semantics, even when we introduce casts, then we must take the second option and type `x` as `Object[]`.

Fortunately the required changes to JIMPLELOCALTYPEINFERENCE are minor. We simply replace lines 9 and 10 of Algorithm 3 with

```
($lhs := e$) : $worklist \Leftarrow worklist(\sigma)$
if $lhs$ matches $v[i]$ then
    $v[i] \Leftarrow lhs$
    $t' \Leftarrow \text{eval}(\sigma, e)[]$
else
    $v \Leftarrow lhs$
    $t' \Leftarrow \text{eval}(\sigma, e)$
end if
```

4.3 Primitive Types

The Java primitive types offer more difficulties for local type inference, due to Java's awkward handling of the small integer types: `boolean`, `byte`, `char`, `short` and `int`. The problems are particularly specific to Java-related languages and do require some non-trivial solutions. In this section I discuss the problems and my solutions in some detail, but to offer an overview:

- there exists verifiable bytecode for which no Jimple equivalent exists, even if we allow semantics-preserving casts;

- and for some assignments there is no single type t such that the assignment is valid if and only if the left-hand-side is typed as an ancestor of t.

Java bytecode makes no distinction between the small integer types. At the bytecode level they are all treated as `int` and can be used interchangeably, though as I discuss later in this section the semantics are sometimes dubious. There are no implicit conversions allowed between the other primitive types `long`, `float` and `double`. These rules are represented by the hierarchy shown in Figure 5a.

Java source allows some implicit 'widening' primitive conversions between the numeric types, but no implicit conversions to or from the `boolean` type. This hierarchy is shown in Figure 5b.

As we might expect, the Jimple type hierarchy for primitives, shown in Figure 5c, lies somewhere between Java bytecode and Java source. There are some implicit widening conversions allowed between small integer types but not `boolean`. Explicit conversions in the form of casts are allowed between any primitive types. Narrowing casts between the small integer types are compiled to bytecode as the integer truncation instructions such at `i2s`, `i2c` and `i2b`. Casts between `boolean` and any small integer type are ignored in the Jimple to bytecode compilation.

The JIMPLELOCALTYPEINFERENCE algorithm (Algorithm 4) is capable of typing all local variables in Jimple, except those that hold small integer values. To make it do this we can type against the bytecode hierarchy rather than the Jimple hierarchy. We can verify that the type of a local variable v depends on the type of any small integer variable only if v is also a small integer variable. So we can use this algorithm to find a least valid typing, requiring the insertion of fewest casts, but with all small integer variables typed as `int`. This typing would not be valid under the Jimple hierarchy, but it does mean that a second stage can ignore all other variables, and only needs to worry about small integers. This section deals with this second stage, where we begin with a least valid typing under the bytecode hierarchy. In this stage we reconsider small integers and find a least valid typing under the Jimple hierarchy. Gagnon *et al.* [2] also divide the problem into the same two stages, which allows convenient comparison in later experiments.

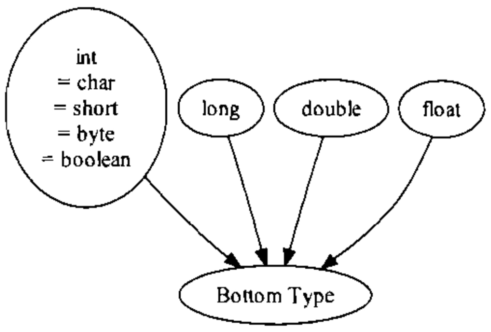
4.3.1 Inserting Small Integer Casts

This difference between the bytecode and Jimple hierarchies leads to the first point of concern in typing primitives. There exist conversions, such as `int` to `byte` and `boolean` to `int`, which are implicit in Java bytecode but must be made explicit by inserting casts in Jimple. Without casts an assignment-valid typing may not even exist for small integer types where the same variable is assigned both a `boolean` and any other small integer type! As mentioned above, some of these casts have the effect of truncating the value of the variable, which may clearly change the program semantics. However an important observation made by Gagnon *et al.* [2] is that if all variables are 'big enough' to hold the values of all types that are ever assigned to them, then any required casts will only affect program semantics when the semantics were dubious (at run-time) to begin with. By *dubious semantics* Gagnon means using a small integer variable with a value greater than expected, based on the static type information in Java bytecode. Such static type information is available for variable uses in method invocations, field assignments and return statements. So dubious semantics can occur at any of these sites. An example code snippet always exhibits dubious semantics:

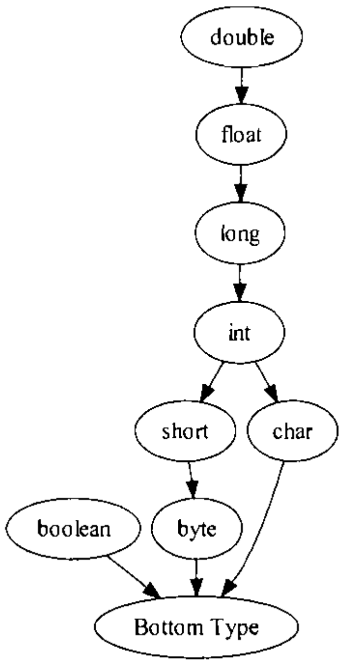
```
1   <untyped> x;
2   x = 5;
3   takesABoolean(x);
4   takesAnInt(x);
```

Line 3 is well-typed under the bytecode hierarchy, even though a small integer value of 5 is passed to the `takesABoolean(boolean)` method, which expects a `boolean`. This is 'dubious' because, without examining the code, we cannot determine whether `takesABoolean(5)` behaves like `takesABoolean(1)` or `takesABoolean(0)`, or maybe differently from both! It is not line 3 itself that is dubious, it is the run-time event where `takesABoolean` is called with a parameter value not equal to 0 or 1. Dubious semantics never occur on line 4 because the `int` type contains value 5.

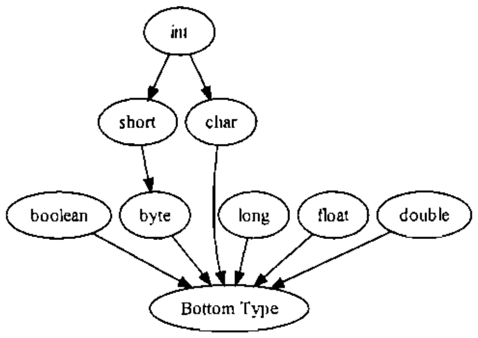
There is no typing for the example given above that is valid under the Jimple hierarchy. If we choose to type `x` as `byte` then lines 2 and 4 are valid but a cast is required on line 3. Similarly if we choose `boolean` then line 3 is valid but lines 2 and 4 require casts. In the their Jimple type inference



(a) The Java Bytecode hierarchy

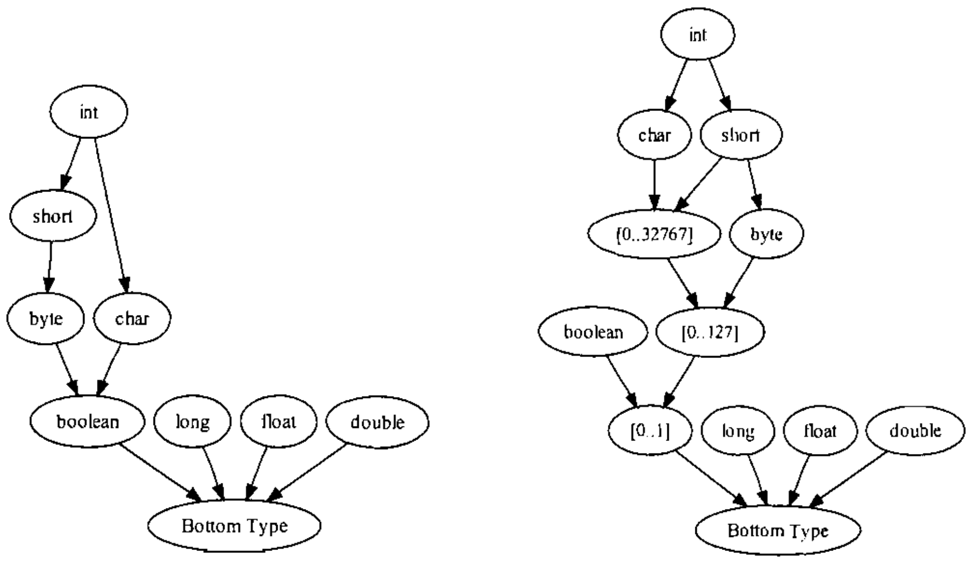


(b) The Java hierarchy



(c) The Jimple hierarchy

Figure 5: Primitive type hierarchies in different languages



(a) The value set hierarchy

(b) The augmented Jimple hierarchy

Figure 6: Modified type hierarchies used by my algorithm

algorithm [2], Gagnon *et al.* accept that the semantics of the original program may be altered by inserting casts, but only where the use would be dubious to begin with. So in the example above we type `x` as `byte` and insert the cast at line 3.

I use the same idea and allow casts where required, after ensuring that the typing maps all small integer variables to a least type that is 'big enough' to hold all values that could be assigned. But what does 'big enough' mean? I write that, for two small integer types t_1 and t_2, $t_1 \subseteq t_2$ if and only if all values of type t_1 are also values of t_2. Now we can easily check that

- for all small integer types t_1, t_2 except `boolean`, $t_1 \subseteq t_2 \equiv t_1 \leq t_2$,
- for all small integer types t, `boolean` $\subseteq t$,
- and for all small integer types t, `boolean` $\subseteq t \equiv t =$ `boolean`.

This partial order can be represented by the *value-set hierarchy* shown in Figure 6a. We can verify that when $t_1 \subseteq t_2$, although a cast may be required if t_2 is used where t_1 is expected, this cast will not change (run-time) semantics unless the semantics of the use would be dubious anyway. I write that an assignment ($t := e$) is value-set-valid under a typing σ if and only if eval$(\sigma, e) \subseteq t$. This is a principle equivalent to assignment-validity but, since the $\subseteq$ relation is not the same as $\leq$, assignment-validity and value-set-validity are not the same. To see the difference consider the example below:

```
1   <untyped> x;
2   x = returnsAnInt();
3   x = returnsABoolean();
```

This is valid Jimple but there is no assignment-valid typing. If we type `x` as `boolean` then line 2 is not assignment-valid, but if we type `x` as any other small integer type then line 3 is not assignment-valid. However we can type `x` as `int` and all assignments are value-set-valid. This suggests a strategy for typing small integers, which guarantees a valid typing requiring the insertion of fewest casts. We find the least value-set-valid typings under the Jimple hierarchy, and then select the typing that requires the fewest number of casts.

But how do we find the set of value-set-valid typings that are least under the Jimple hierarchy? First we apply Algorithm APPLYASSIGNMENTCONSTRAINTS *against the value-set hierarchy*! This gives us the set Σ that is least under the value-set hierarchy, but not under the Jimple hierarchy. But notice that Σ is never empty, because typing all small integer variables as `int` is always value-set valid. Now any typing $\sigma \in \Sigma$ must also be least under the Jimple hierarchy. So we siuply need to find the set of least typings (under $\leq$) from the set $\{\sigma' | \exists\sigma \in \Sigma, \sigma \subseteq \sigma'\}$.

By inspecting the hierarchies we can make an important observation. Suppose that for some typings σ_1, σ_2 we know $\sigma_1 \subseteq \sigma_2$. Now there always exists a σ_1' that is a *boolean-extension* of σ_1 such that $\sigma_1' \leq \sigma_2$. I define a *boolean-extension* of a typing σ as any typing equal to σ except that variables mapping to `boolean` nnder σ *may* also map to `char` or `byte` under a boolean-extension of σ. For example, the set of boolean extensions of the typing {`x : boolean, y : short, z : boolean`} is

{`x : boolean, y : sbort, z : boolean`},	{`x : byte, y : sbort, z : boolean`},	{`x : char, y : short, z : boolean`},
{`x : boolean, y : sbort, z : byte`},	{`x : byte, y : sbort, z : byte`},	{`x : char, y : short, z : byte`},
{`x : boolean, y : short, z : char`},	{`x : byte, y : sbort, z : char`},	{`x : cbar, y : short, z : char`}

Algorithm 5 FINDBOOLEANEXTENSIONS(σ)

```
1:  $\Sigma \Leftarrow \{\sigma\}$
2:  for all local variables $v$ do
3:    if $\sigma(v)$ = boolean then
4:      for all $t \in$ {byte, char} do
5:        $\sigma' \Leftarrow \sigma[v \mapsto t]$
6:        $\{\sigma'\} \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma'$) {Using the value-set hierarchy! For a different
          reason we shall see that version 3 is required in Jimple, which is introduced in the next
          section.}
7:        $\Sigma \Leftarrow \Sigma \cup$ FINDBOOLEANEXTENSIONS($\sigma'$)
8:      end for
9:    end if
10: end for
11: return $\Sigma$
```

As FINDBOOLEANEXTENSIONS (Alogrithm 5) I present a pseudo-code algorithm for finding the boolean-extensions of a typing σ. Notice that whenever we change a type we re-execute APPLYASSIGNMENTCONSTRAINTS (under the value-set hierarchy) to account for the possible changes to expression types in the method.

Now I am ready to present JIMPLESMALLINTEGERLOCALTYPEINFERENCE (Algorithm 6), an algorithm for finding the least (under the Jimple hierarchy $\leq$) value-set-valid typing, which requires the insertion of fewest casts. We first we find the set Σ' of value-set-valid typings that is least under the value-set hierarchy ($\subseteq$). We then take the union Σ of all boolean-extensions of all typings in Σ'. We finally minimize Σ under the Jimple hierarchy and return tbis. Now we know that any casts required by these value-set-valid typings will be acceptable, only changing program semantics if they are already dubious. So I select any typing from Σ that requires the insertion of fewest casts.

So we should be able to perform complete local type inference in Jimple by first using JIMPLELOCALTYPEINFERENCE (Algorithm 4) and then JIMPLESMALLINTEGERLOCALTYPEINFERENCE (Algorithm 6.)

4.3.2 The Type of Short Integer Constants

But there is another big problem! Thus far APPLYASSIGNMENTCONSTRAINTS has relied upon a well-defined eval : $\Sigma \times E \mapsto T$ function, which doesn't exist for Jimple! The perpetrator for this is the humble *small integer constant* expression. Consider the simple assignment

```
<untyped> x;
x = 5;
```

Clearly local variable `x` could be typed as `byte` or `char`, and both such typings are both least assignment-valid and least value-set valid typings. So what is the single value of eval$(\{x \mapsto \bot\}, 5)$?

Algorithm 6 JIMPLESMALLINTEGERLOCALTYPEINFERENCE(σ)
Local type inference for Jimple small integers

```
Require: $\sigma$ is a least valid typing for all variables except small integers, requiring the insertion of fewest casts. All small integers are typed as int.
Ensure: The return valne is a least valid typing for all local variables, requiring the insertion of fewest casts.
1:  for all local variables $v$ do
2:    if $\sigma(v)$ = int then
3:      $\sigma(v) \Leftarrow \bot$
4:    end if
5:  end for
6:  $\Sigma' \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma$) {Using the value-set hierarchy! For a different reason we shall see that version 3 is required in Jimple, which is introduced in the next section.}
7:  $\Sigma \Leftarrow \emptyset$
8:  for all $\sigma \in \Sigma'$ do
9:    $\Sigma \Leftarrow \Sigma \cup$ FINDBOOLEANEXTENSIONS($\sigma$)
10: end for
11: $minCasts \Leftarrow \min\{$COUNTCASTSREQUIRED$(\sigma) | \sigma \in \Sigma\}$
12: $\sigma \Leftarrow$ any element of $\Sigma$ where COUNTCASTSREQUIRED($\sigma$) $= minCasts$
13: Insert casts to make $\sigma$ use valid
14: return $\sigma$
```

Suppose eval$(\{x \mapsto \bot\}, 5) =$ byte, then by the definition of assignment-validity (5), byte $\not\leq$ char would imply that line 1 is not assignment-valid under $\{x \mapsto$ char$\}$, which is not a correct model of Jimple. Of course this same argument would apply to any single-valued eval fnnction. So either we need a different method or a different hierarchy! I consider both of these approaches.

First in Section 4.3.3 I propose a generalized version of APPLYASSIGNMENTCONSTRAINTS to support a multi-valued eval function. This algorithm is equally efficient as earlier versions in languages without a multi-valued eval function. However my experiments showed small integer constants are common in typical Jimple, resultiug in the generation of many candidate typings. In Section 4.3.4 I propose a 'type promotion' algorithm for the small integer types, which uses a specially augmented type hierarchy to enable a single-valued eval function. This is a very different algorithm for small integer typing that does not rely on the value-set hierarchy, but consequently cannot guarantee to generate a typing requiring the insertion of fewest casts. But if it can find a typing, which it can in almost all typical cases, then that typing is guaranteed to be minimal. Finally in Section 4.3.5 I combine the two approaches to give an algorithm that uses type promotion in almost all cases, but reverts to the slower method when the insertion of casts is required.

4.3.3 Supporting a Multi-Valued eval Function

Perhaps the most obvious solution would be to generalize eval : $\Sigma \times E \rightarrow 2^T$. Intuitively this means that expression e can be converted to all ancestors of each of the types in eval(σ, e). We can adapt the formal notion of assignment-validity: an assignment $v := e$ is *valid* under a typing σ if and only if $\exists t \in$ eval$(\sigma, e), t \leq \sigma(v)$.

The proof of Section 3.0.2 can be generalized by changiug the definition of *step* (8) to

$$step(\sigma, \sigma') = \forall(v := e) \in A, \exists t \in \text{eval}(\sigma, e), t \leq \sigma'(v)$$

This leads to a modification of APPLYASSIGNMENTCONSTRAINTS (Algorithm 7.)

Notice the extra loop added on line 10, around the loop introduced in Section 3 to handle a multi-valued lca function. The similarity of the two generalizations is clear. Remember that when we introduced this first loop, we relied on the infrequency of lca giving multiple values, since the algorithm has the potential for exponential blow-up. Unfortunately, in Jimple at least, assigning small integer constants to local variables is very common, so it is not unusual for the eval function to give several values. Indeed, experiments showed that Algorithm 7 performs much worse than the integer typing algorithm of Gagnon

Algorithm 7 ApplyAssignmentConstraints(σ) Version 3
Local type inference supporting a multi-valued eval function

```
1:  for all local variables $v$ do
2:      $v \Leftarrow \bot$
3:  end for
4:  $\Sigma \Leftarrow \{\sigma\}$
5:  $worklist(\sigma) \Leftarrow$ all assignments
6:  while $\exists \sigma \in \Sigma, worklist(\sigma) \neq \emptyset$ do
7:      Pick $\sigma \in \Sigma, worklist(\sigma) \neq \emptyset$
8:      $\Sigma \Leftarrow \Sigma \setminus \{\sigma\}$
9:      $(v := e) : worklist \Leftarrow worklist(\sigma)$
10:     for all $t' \in \text{eval}(\sigma, e)$ do
11:         for all $t$ in $\text{lca}(\sigma(v), t')$ do
12:             if $t = \sigma(v)$ then
13:                 $\Sigma \Leftarrow \Sigma \cup \{\sigma\}$
14:             else
15:                 $\sigma' \Leftarrow \sigma[v \mapsto t]$
16:                 $worklist(\sigma') \Leftarrow worklist(\sigma) +\!\!+ \text{depends}(v)$
17:                 $\Sigma \Leftarrow \Sigma \cup \{\sigma'\}$
18:             end if
19:         end for
20:     end for
21: end while
22: minimize $\Sigma$
23: return $\Sigma$
```

et al. [2], although this is not strictly a fair comparison, since their algorithm does not guarantee the insertion of as few casts as possible.

4.3.4 A Type Promotion Algorithm

Remember we are tackling local type inference in two stages: firstly we treat all small integer types as the same type, and then we need only consider small integer types in the second stage. This two-stage strategy is also employed by Gagnon *et al.* [2]. For their second stage they augment the Jimple hierarchy by inserting types [0..1], [0..127] and [0..32767] as shown in Figure 6b. I call these new types *imaginary* types. Having augmented the hierarchy, Gagnon *et al.* then use a method very similar to their main algorithm, which I have summarized in Section 6. I choose to augment the hierarchy in the same way, but otherwise their algorithm is very different to what I develop in this section.

The most obvious benefit of augmenting the primitive type hierarchy is that it allows a single-valued eval function. The part of the eval function describing iuteger constants can be defined as shown in Algorithm 8.

Now we can use any version of ApplyAssignmentConstraints to find the least assignment-valid typing for small integer variables under the augmented hierarchy. Under the augmented hierarchy the lca and eval functions are always single-valued. So after this step we have only found an assignment-valid typing. Clearly, before we can return the typing, we must promote the imaginary types to Jimple types. This promotion is achieved by examining local variable uses.

There is another important feature of the augmented hierarchy: a suitable *promote* : $T^2 \mapsto T$ function exists. It is this that doesn't exist for the entire Jimple type hierarchy, which is why this type promotion algorithm only works for small integers. The only crucial property of the *promote* function is that for any $t_{low} \leq t_{high}, \forall t \in T, (t_{low} < t \leq t_{high} \implies promote(t_{low}, t_{high}) \leq t)$ This property has the following intuitive meaning:

> If we know some lower bound type t_{low} for a variable v, we know that v can never be typed as t_{low}, and we know some upper bound type t_{high} for v, then $promote(t_{low}, t_{high})$ is also a lower bound type for the variable.

Algorithm 8 eval(_, IntConstant(i))

```
if $0 \le i \le 1$ then
    return [0..1]
else if $2 \le i \le 127$ then
    return [0..127]
else if $128 \le i \le 32767$ then
    return [0..32767]
else if $32768 \le i \le 65535$ then
    return char
else if $-128 \le i \le -1$ then
    return byte
else if $-32768 \le i \le -129$ then
    return short
else
    return int
end if
```

Remember we always 'know that v can never be typed as' any imaginary type. The *promote* function is the key to what I have called the the type promotion algorithm for small integers. I provide the pseudo-code as TYPEPROMOTION (Algorithm 9.) Here we start with an assignment-valid typing under the augmented Jimple hierarchy. We then begin iterating through every local variable with an imaginary type. For each such variable we consider every use, and using the *promote* function try to promote the type toward a Jimple type, accounting for the constraints imposed by the nses. If ever we reach a stage where the typing is not use-valid then we fail, since this tells us that casts will certainly be required. If, after considering all uses for some variable, the type of the variable has changed, then we re-apply APPLYASSIGNMENTCONSTRAINTS using the augmented hierarchy to account for the effect this change may have on the least assignment-valid typing.

After having fouud a valid typing this may still map some local variables to iruaginary types. In this case we can choose any least non-imaginary supertype of the imaginary type. I simply apply a fixed promotion.

4.3.5 A Final Algorithm

So defining a multi-valued *eval* function and using JIMPLESMALLINTEGERLOCALTYPEINFERENCE (Algorithm 6) is slow. TYPEPROMOTION (Algorithm 9) is fast, but cannot always provide a typing. My solution is to use type promotiou wherever possible, and then revert to JIMPLESMALLINTEGERLOCALTYPEINFERENCE in the rare cases that casts are required on small integer variables. This is guaranteed to give the least valid typing haviug inserted the fewest casts. I present a pseudo-code description of this algorithm as JIMPLELOCALTYPEINFERENCE version 2 (Algorithm 10.)

And this concludes the algorithm development section! JIMPLELOCALTYPEINFERENCE is at last capable of finding the least valid typing for all Jimple code, requiring the insertion of fewest casts.

5 Experimental Evaluation

For all experiments I iuplemented my algorithm as a drop-in replacement for the existing type inference algorithm which forms part of the Soot framework [7]. This original algorithm is due to Gagnon *et al.* [2], and provides a basis for performance comparison both in terms of execution time as well as tightness of the typings generated. The algorithms are invoked as part of the bytecode to Jimple translation.

I identified, with recommendations from the co-authors of the paper, a number of bytecode packages to be used as , comprising 295598 methods in total. It was crucial that the bytecode tested was not all compiled from Java source. Other bytecode sources may make relative use of different language features such as multiple inheritance. The benchmarks chosen included bytecode compiled from the Java, Groovy, Scheme and Scala languages. The benchmarks are listed in Figure 7.

Algorithm 9 TYPEPROMOTION(σ
Type promotion algorithm for small integer types

Require: σ is a least valid typing for all variables except small integers, requiring the insertion of fewest casts. All small integers are typed as `int`.
Ensure: The return value is a least valid typing for all local variables, without inserting additional casts. This algorithm fails if and only if one or more additional casts (between small integer types) are required.

```
 1: for all local variables $v$ do
 2:   if $\sigma(v)$ = int then
 3:     $\sigma(v) \Leftarrow \perp$
 4:   end if
 5: end for
 6: $\{\sigma\} \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma$) {Using the augmented Jimple hierarchy! Any version
    will do, the return value is always a singleton set.}
 7: for all uses $(v, t) \in U$ where $\sigma(v) \leq$ int do
 8:   if $\sigma(v) \not\leq t$ then
 9:     $\sigma$ is not use-valid; fail
10:   end if
11:   if $\sigma(v) \in$ { [0..1], [0..127], [0..32767] } then
12:     $\sigma(v) \Leftarrow promote(\sigma(v), t)$
13:   end if
14: end for
15: for all local variables $v$ do
16:   if $\sigma(v)$ = [0..1] then
17:     $\sigma(v) \Leftarrow$ boolean
18:   else if $\sigma(v)$ = [0..127] then
19:     $\sigma(v) \Leftarrow$ byte
20:   else if $\sigma(v)$ = [0..32767] then
21:     $\sigma(v) \Leftarrow$ char
22:   end if
23: end for
24: return $\sigma$
```

Algorithm 10 JIMPLELOCALTYPEINFERENCE Version 2
Complete local type inference for Jimple

```
 1: $\Sigma \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma_\perp$) {Using the bytecode type hierarchy!}
 2: $minCasts \Leftarrow \min\{$COUNTCASTSREQUIRED$(\sigma) | \sigma \in \Sigma\}$
 3: if $minCasts > 0$ then
 4:   Apply stage A transformation
 5:   $\Sigma \Leftarrow$ APPLYASSIGNMENTCONSTRAINTS($\sigma_\perp$) {Using the bytecode type hierarchy!}
 6:   $minCasts \Leftarrow \min\{$COUNTCASTSREQUIRED$(\sigma) | \sigma \in \Sigma\}$
 7: end if
 8: $\sigma \Leftarrow$ any element of $\Sigma$ where $countCasts(\sigma) = minCasts$
 9: Insert casts to make $\sigma$) use valid under the bytecode type hierarchy
10: if $\sigma \Leftarrow$ TYPEPROMOTION($\sigma$ fails then
11:   $\sigma \Leftarrow$ JIMPLESMALLINTEGERLOCALTYPEINFERENCE($\sigma$)
12: end if
13: return $\sigma$
```

All experiments were carried out with the rigor required for the conference paper. Each method in each benchmark was tested independently 5 times using both the algorithm of Gagnon and my algorithm. For each method the times taken were recorded and the typings generated were compared. The mean of the middle 3 timings for each algorithm was taken. All experiments were run on the same quad-core Intel Xeon 3.2GHz compnter with 4GB RAM running Linux 2.6.8 SMP and the Sun compiler and VM version 1.5.

I present experiments in two sets A and B. In set A I evaluate the performance of the general type inference algorithm, without the extensions to support Jimple small integer types. In set B I evaluate the performance of the small integer typing stage separately. The motivation for the separation is that the results in set A are interesting to all applications of local type inference, whereas the experiments in set B only evaluate a Jimple-specific enhancement.

5.1 Experiment Set A

First of all I present experiments on JIMPLELOCALTYPEINFERENCE Version 1 (Algorithm 4) against the bytecode type hierarchy. This supports multiple inheritance, fixing untypable methods and assignments to arrays, bnt considers all small integer types to be the same. Gagnon's algorithm is implemented in snch a way that it is easy to disable the secondary small integer typing step completely, and all small integer variables are also typed as `int`. So both algorithms are doing an equivalent 'amonnt of work' and the comparison is fair.

Table 1 summarizes the results of experiment set A. I aggregate the individual results for each method into summary values for each benchmark, and then finally a summary over all benchmarks. Shown are the number of methods in each benchmark, the total (middle mean) time spent assigning types using Gagnon's algorithm, and the corresponding total for my algorithm. From these two times I determine the relative improvement. The typings generated by each algorithm are compared (small integer variables are ignored in this comparison) and I record the number of methods for which my algorithm found a tighter typing. Of course, the optimality of my algorithm has been proved, so it never generates a weaker typing. To obtain some indication of the extent of multiple inheritance in each benchmark I record the mean number of candidates generated by APPLYASSIGNMENTCONSTRAINTS. Finally I record the number of methods successfully typed after each of the stages of transformations discussed in Section 4.1. Gagnon's algorithm uses these same stages, indeed he introduced them in [2], so each method is typed at the same stage in both my algorithm and Gagnon's.

Benchmark	# Methods	Old Time (s)	New Time (s)	Improvement	# Tighter	Mean # Cndts.	# No Trans.	# Stg. A	# Stg. B
`rt`	107792	84.48	10.77	7.84x	39	1.00032	107681	77	34
`tools`	14180	13.07	2.37	5.52x	5	1.00014	14160	17	3
`abc-complete`	33866	480.28	4.37	109.88x	52	1.00027	33865	1	0
`jython`	9192	6.67	1.25	5.35x	0	1.00000	9187	5	0
`groovy`	13799	10.12	1.92	5.27x	7	1.00087	13778	4	17
`gant`	707	1.75	0.44	4.01x	0	1.00000	702	3	2
`kawa`	9226	7.70	1.58	4.88x	25	1.00618	9195	31	0
`scala`	65161	37.66	5.36	7.03x	117	1.00453	64865	296	0
`cso`	2395	2.20	0.51	4.34x	6	1.00167	2392	3	0
`jigsaw`	13577	11.50	1.84	6.24x	1	1.00007	13571	6	0
`jedit`	5980	5.74	1.09	5.26x	12	1.00318	5969	0	11
`bluej`	5690	5.37	0.98	5.48x	0	1.00053	5690	0	0
`java3d`	13453	17.86	2.46	7.26x	5	1.00037	13453	0	0
`jgf`	557	3.61	0.48	7.59x	0	1.00000	557	0	0
`havoc`	23	198.53	0.46	428.17x	0	1.00000	23	0	0
Total	295598	886.53	35.87	24.72x	269		295088	443	67

Table 1: Performance comparison between both algorithms without typing of small integer variables

From the results is is clear that my algorithm shows a typical improvement of around 6 times, but in two cases `abc-complete` and `havoc` this improvement is much greater. I examined these benchmarks and found that they each contain several huge (>9000 Jimple statements) methods. This suggests that my algorithm might show bigger improvements with increasing method size, so I investigate this further.

Figure 8 plots a point for of every one of the 295598 methods tested at the time spent inferring types against method length (number of Jimple statements.) A cube-root scaling has been chosen for the y-axis, and this highlights that the theoretical polynomial complexity [2] of Gagnon's algorithm is achieved in practice. My algorithm has a theoretical exponential complexity, but we can see that in practice it is usually somewhat better than cubic.

Fignre 9 shows a similar plot, but this time containing only the points for my algorithm on a linear y-axis. The apparent straight line of points indicates a linear complexity. Of course this is a common-case and not a worst-case complexity, since there are a significant number of points above this line.

5.2 Experiment Set B

For these experiments I use the final JimpleLocalTypeInference Version 2 (Algorithm 10). I also enable the small integer typing stage of Gagnon's algorithm. So the comparison is between complete local type inference algorithms for Jimple.

Benchmark	Soot Time (s)	Soot % total	My Time (s)	My % total	Integer Imprv.	Total Imprv.	# Tighter
rt	22.88	21.31	8.98	45.46	2.55x	5.44x	1061
tools	5.51	29.66	1.04	30.50	5.30x	5.45x	165
abc-complete	89.98	15.78	2.90	39.88	31.03x	78.43x	177
jython	3.59	35.02	1.13	47.50	3.19x	4.32x	45
groovy	3.84	27.50	1.39	41.94	2.77x	4.22x	386
gant	0.59	25.06	0.39	47.30	1.49x	2.82x	70
kawa	3.42	30.74	0.78	33.01	4.40x	4.72x	198
scala	6.28	14.30	2.56	32.30	2.46x	5.55x	190
cso	0.43	16.23	0.31	38.02	1.37x	3.21x	6
jigsaw	3.20	21.77	1.07	36.62	3.00x	5.05x	131
jedit	2.17	27.43	0.56	34.09	3.84x	4.78x	170
bluej	1.12	17.28	0.46	31.88	2.44x	4.51x	78
java3d	5.60	23.89	1.58	39.15	3.54x	5.80x	264
jgf	0.81	18.36	0.24	33.90	3.33x	6.14x	16
havoc	239.55	54.68	0.41	47.03	581.90x	500.47x	0
Total	388.98	30.50	23.79	39.88	16.35x	21.38x	2957

Benchmark	# My Stg. 1	# My Stg. 2	# Soot Stg. 1	# Soot Stg. 2
rt	107792	0	107792	0
tools	14180	0	14180	0
abc-complete	33866	0	33866	0
jython	9192	0	9192	0
groovy	13799	0	13799	0
gant	707	0	707	0
kawa	9202	24	9223	3
scala	65160	1	65161	0
cso	2395	0	2395	0
jigsaw	13577	0	13577	0
jedit	5980	0	5980	0
bluej	5690	0	5690	0
java3d	13453	0	13453	0
jgf	557	0	557	0
havoc	23	0	23	0
Total	295573	25	295595	3

Table 2: Performance comparison between both algorithms for typing of small integer variables

The results for set B are shown iu Table 2. For each algorithm I record the execution time of the small integer typing stage, aud also the proportion of the total type inference time this represents. I then give relative improvements of my algorithm for small integer typing alone as well as complete type inference. I also count the number of tighter typings found by my algorithm, though this number is not as meaningful as it may at first appear, since Gagnon's equivalent of an eval function does not always provide the tightest type possible for expressions involving small integers. Finally I record the number of methods typed at each of the two stages of small integer typing in both algorithms. My stage 1 is the TypePromotion algorithm and my stage 2 is the slower JimpleSmallIntegerLocalTypeInference algorithm that is only used when the insertion of small integer casts are required. Soot also uses two stages for integer typing, but these are not comparable to mine.

My first table of results shows that both algorithms spend a significant amount of execution time typing small integer variables. The Soot algorithm spends around 20% and my algorithm spends around

30%. These comparisons show a similar, but slightly lesser typical improvement of 2 to 4 times for the small integer typing stage alone. The overall improvement for the whole Jimple type inference algorithm is 4 to 5 times.

From the second table we notice that all methods are typable using type promotion for small integer types, and very few require inserting small integer casts. These are the Kawa and Scala benchmarks, which both contain code not generated by a Java compiler. Indeed, one might expect code generated by a Java compiler to be typable without requiring cast insertion, unless it had undergone some kind of optimization or obfuscation.

6 Related Work

There has been a considerable amount of work on type inference for object-oriented languages, and it is still an active topic for research. In this section I summarize two closely related papers that present algorithms for the problem of local type inference in languages related to Java.

6.1 Gagnon *et al.*

As stated in the introduction, my initial motivation for this work was to improve the performance of local type inference in the Java bytecode to Jimple procedure of the Soot framework [7]. The original algorithm is due to Gagnon *et al* [2]. I have referenced Gagnon's work a number of times throughout this report, and it has indeed been an extremely useful analysis of the problem. It was, after all, the only type assignment algorithm ever implemented for Jimple. In my comparison below we will see that the core of each algorithm is substantially different, though I do employ several of Gagnon's ideas to support some unusual quirks of Jimple.

First I compare the core of each algorithm, which I have presented in Section 2. This is the algorithm for local type inference supporting multiple inheritance, but where each expression has a single least type, and a valid typing always exists. Here Gagnon's algorithm considers local variable assignments and uses together, and builds a directed constraint graph. Each node takes one of two forms:

- *hard nodes* represent specific Java types, and are written as the type name such as `String` or `double`;
- *soft nodes* represent the type of a local variable and are written as $T(v)$ where v is a local variable.

Edges in the constraint graph represent subtyping constraints. So the edge $a \longleftarrow b$ means that under any valid typing node a will be a subtype of node b. The first part of the algorithm builds the complete constraint graph for all assignments and uses. For example, in the example snippet below, line 2 adds the constraint `String` $\longleftarrow T($`x`$)$ and line 3 adds $T($`x`$) \longleftarrow$ `Object`

```
1  <untyped> x;
2  x = new String("Some String");
3  takesAnObject(x);
```

Having built the constraiut graph it is next 'solved' by applying several rules, which are guaranteed to find a valid typing if it exits. However it is not guaranteed to be minimal. Gagnon's algorithm is clearly carefully designed and there did not seem to be much scope for improving their ideas, thus I chose to begin again from an initial specification of the problem. I managed to avoid constructing a potentially very large constraint graph, since I noticed that most methods could be typed 'by hand' without such a graph.

6.2 Knoblock and Rehof

Knoblock and Rehof [4] present an algorithm for local type inference (which they call type elaboration) on a language called *Java Intermediate Representation (JIR)*. JIR is very similar to Jimple in that it is also a statically typed, stackless representation of Java bytecode. One difference is that JIR is always in Single Static Assignment (SSA) form. This means that all variables are split, by flow analysis, to the extent that they are assigned at one and only one statement in the code. If the same variable can be assigned at different branches of a control flow block then the variable is still split. In these cases a special Φ function may be used after the block to select whichever value was assigned. Jimple undergoes similar variable splitting but not to the extent of reaching SSA form. Jimple allows each variable to be assigned more than once in different branches of control flow statements, so avoiding the need for Φ functions. There are some methods that are untypable in Jimple but would be typable having being converted to SSA form such as the `untypableA` example in Figure 4. My algorithm handles this by applying tbe stage A transformation discussed in 4.1, which is based on the ideas of Gagnon *et al.* [2].

Knoblock and Rehof's algorithm requires that the type hierarchy form a lattice, so each pair of types has a unique least-upper-bound and greatest-lower-bound. As I demonstrated in Section 3 the Java type hierarchy does not form a lattice, which is why I generalized APPLYASSIGNMENTCONSTRAINTS to support the Java hierarchy. Instead Knoblock and Rehof choose to insert additional types where required. This may be acceptable for optimization purposes, as long as the environment allows the program to be changed as a whole. But many analysis and decompilation applications will require the program maintain its original type hierarchy, in which case Knoblock and Rehof's algorithm would be unusable.

It is difficult to compare the performance of my algorithm with theirs. Their experiments show typical linear complexity, as do mine in Figure 9. They do not make available their implementation so I cannot perform rigorous comparisons. It is worth noting that my experiments are somewhat more extensive. I test more than ten times as many methods and my experiments include benchmarks generated from a range of different source code languages. They only test benchmarks compiled from Java source.

7 Conclusions

I have successfully developed a novel local type inference algorithm from an initial specification of the problem. I first provide an intuitive derivation of the algorithm and then offer a formal proof of correctness. I go on to offer generalizations to the algorithm, supporting more language features, and finally achieve local type inference in Jimple.

I have carried out careful experimental evaluation to compare the performance of my algorithm to that of Gagnon *et al.* [2], which is the only implemented alternative for local type inference in Jimple. Experiments showed a typical 4-fold to 5-fold execution time improvement across a wide range of benchmarks, and a much greater increase where very large methods exist. Experiments were not confined to code compiled from Java, and include code compiled from very different languages like Scala and Scheme. My algorithm is proven to always give a tightest possible typing. Experiments show that Gagnon's algorithm rarely but sometimes gives suboptimal typings.

The theoretical worst case complexity of my algorithm is exponential, whereas Gagnon's algorithm is polynomial. But experiments of execution time against method length show a typically linear trend whereas Gagnon's show a cubic trend. My algorithm is very much optimized for type hierarchies in which most pairs of types have a single least-common-ancestor, which probably includes most Java bytecode in existence today. Of course if a language appeared that made much greater use of multiple inheritance then my algorithm may not be appropriate. But as a practical 'workhorse' implementation I believe this is seriously worthy of consideration. And this is supported by the decision of the Soot framework's maintainers to replace their existing type inference algorithm with mine.

7.1 Future Work

The greatest scope for future work is extending the application of my algorithm from local type inference to global type inference. This involves inferring types for method signatures and public fields as well as local variables. The global type inference problem is currently an area of active research, most of which builds upon the work of Palsberg and Schwartzbach [6]. One could begin by treating method parameters, return values and fields in the same way as local variables, and then using my algorithm on the program as a whole.

7.2 Personal Report

Overall I am pleased with the outcome of this project, especially that OOPSLA thought the work worthy of a conference paper. I believe there is no doubt it solves the problem that I began with: to 'speed up local type inference in Jimple'.

My main difficulty was the lack of a formal specification of the Jimple language. I was forced to examine the Soot source code to deduce the typing rules. Particularly useful was the code for the original type assigner that is used in bytecode to Jimple translation, and the Jimple to bytecode compiler (`soot.jimple.JasminClass.java.`).

Haviug completed this project I uow have a much better understanding of the formal type systems in object-oriented programming languages. I have learned techniques for adapting an intuitive derivation of an algorithm into a formal description, and then proving correctness. I have learned how to carry out experiments with the rigor required for serious research. And, perhaps most importantly, I have learned how to plan and write a research paper as part of a small group.

References

[1] Eric Bodden. A denial-of-service attack on the Java bytecode verifier. `http://www.bodden.de/research/javados/`, 2008.

[2] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[3] Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, 6:133–134, 1928.

[4] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for java bytecode. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–242, New York, NY, USA, 2000. ACM.

[5] Tim Lindholm and Franl Yellin. *Java Virtual Machine Specification, Second Edition.* Prentice Hall, 1999.

[6] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY. 1991. ACM Press.

[7] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

`rt` is the Snn Java 1.5 runtime library. The main interest of this benchmark is its size (108K methods), and the fact that it exercises many features of the Java language.

`tools` is the Sun JDK 1.5 tools library including *javac*. Again this benchmark is chosen because it is an interesting piece of Java, albeit of modest size (14K methods.)

`abc-complete` is version 1.2.1 of the AspectBench Compiler for the AspectJ programming language including Soot and Polyglot. This is interesting as a benchmark because it contains many large, generated methods.

`jython` is version 2.2.1 of Jython, a Python implementation written in Java. This is chosen as a mid-sized example of typical Java code.

`groovy` is version 1.5.4 of the compiler for the Groovy programming language. Again this is written in Java and provides another benchmark containing typical Java code.

`gant` is version 1.1.1 of the Gant build system, similar to Ant but compiled to bytecode by Groovy instead of *javac*. This is an important experiment because the algorithm is designed to handle all valid bytecode, not just bytecode generated from Java source.

`kawa` is version 1.9.1 of the Kawa compiler for the Scheme programming language. Here part of the jar is bootstrapped, again giving bytecode sequences that would not normally occur as the output of *javac*.

`scala` is version 2.7.0 of the Scala compiler and runtime library, both of which are written in Scala, and compiled by the Scala compiler (again, instead of *javac*).

`cso` is a concurrency library, loosely inspired by the CSP calculus, written by Bernard Sufrin in Scala. This benchmark is also compiled by Scala instead of *javac*.

`jigsaw` is version 2.2.6 of the W3C's Jigsaw web server implementation, and this is included as a typical web application written in Java.

`jedit` is version 1.4pre13 of the jEdit text editor, an example of an interactive application written in Java.

`bluej` is version 2.2.1 of the BlueJ IDE for the Java programming language, again chosen as an interactive applicatiou.

`java3d` is version 1.5.1 of the Java 3D API. As we shall see later in this paper, (numerical) primitive types can pose a challenge for type inference algorithms on Java bytecode, and this is a potential example of that phenomenon.

`jgf` is version 2.0 of the Java Grande Forum Sequential Benchmark Suite, again chosen for its use of primitive type operations.

`havoc` is a contrived example of Java bytecode which takes unusually long for the JVM to verify, in fact it was designed [1] to be a denial-of-service attack on the Java bytecode verifier.

Figure 7: Experiment Benchmark Descriptions

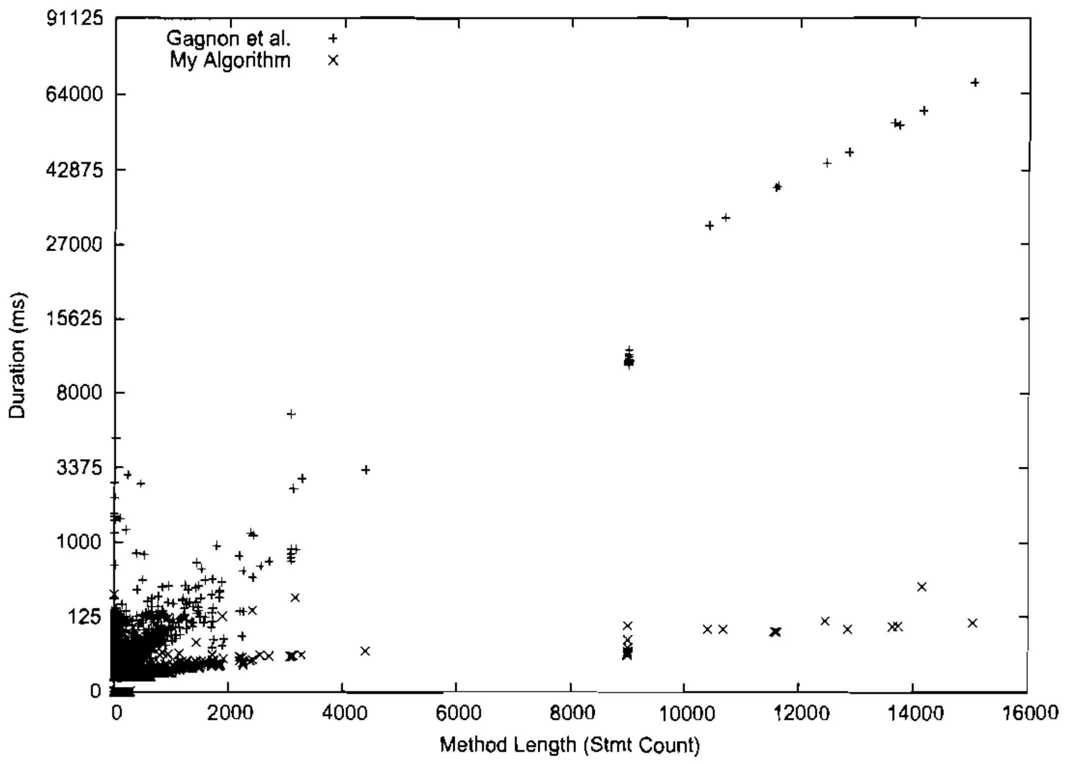


Figure 8: Comparison of the two algorithms: ruutime against method size; cube-root plot

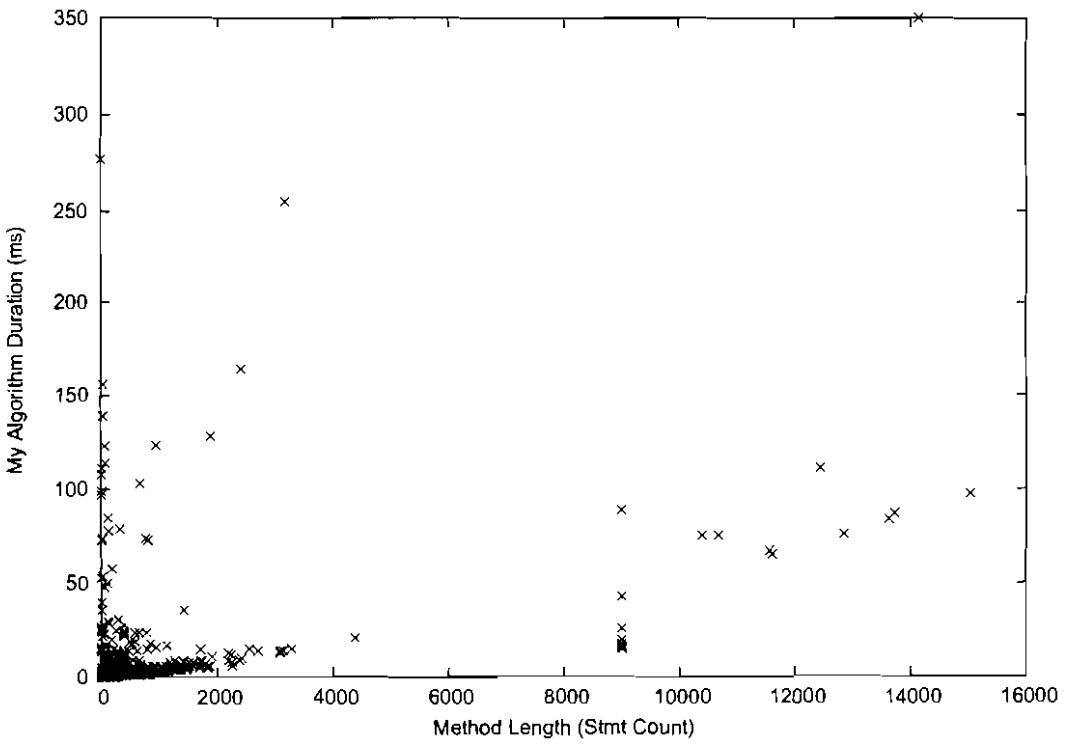


Figure 9: My algorithm: runtime against method size

Efficient Local Type Inference

Ben Bellamy Pavel Avgustinov Oege de Moor Damien Sereni
Programming Tools Group, University of Oxford, UK
benjamin.bellamy@magd.ox.ac.uk, {pavel,oege,damien}@comlab.ox.ac.uk

Abstract

Inference of static types for local variables in Java bytecode is the first step of any serious tool that manipulates bytecode, be it for decompilation, transformation or analysis. It is important, therefore, to perform that step as accurately and efficiently as possible. Previous work has sought to give solutions with good worst-case complexity.

We present a novel algorithm, which is optimised for the common case rather than worst-case performance. It works by first finding a set of minimal typings that are valid for all assignments, and then checking whether these minimal typings satisfy all uses. Unlike previous algorithms, it does not explicitly build a data structure of type constraints, and it is easy to implement efficiently. We prove that the algorithm produces a typing that is both sound (obeying the rules of the language) and as tight as possible.

We then go on to present extensive experiments, comparing the results of the new algorithm against the previously best known method. The experiments include bytecode that is generated in other ways than compilation of Java source. The new algorithm is always faster, typically by a factor 6, but on some real benchmarks the gain is as high as a factor of 92. Furthermore, whereas that previous method is sometimes suboptimal, our algorithm always returns a tightest possible type.

We also discuss in detail how we handle primitive types, which is a difficult issue due to the discrepancy in their treatment between Java bytecode and Java source. For the application to decompilation, however, it is very important to handle this correctly.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Experimentation, Languages, Performance

Keywords type inference, program analysis

1. INTRODUCTION

We discuss *local type inference*: the problem of inferring static types for local variables in an object-oriented language. We assume the types of method signatures and fields are given but type information for local variables is unavailable; this is precisely the case with Java bytecode, in which method calls are fully resolved and fields are typed but local variables have been "compiled away" into stack code. We then wish to compute types for local variables that are as tight as possible, in the sense that they are as low in the inheritance hierarchy as the typing rules allow.

Motivation The motivating application is the conversion of Java bytecode to a typed 3-address intermediate representation for analysis, transformation and decompilation [8, 15]. At first it might seem trivial to infer types for locals from bytecode, but this is not so because in bytecode, stack locations are given types depending on the control flow. By contrast, we wish to infer static types that are not flow-sensitive. Gagnon *et al.* [8] investigated this problem in depth, and presented an algorithm that has good worst-case complexity, which is at the heart of the popular Soot framework [26]. In certain cases, however, that algorithm performs quite badly. For example, when processing the *abc* compiler [1] with itself, 98% of the time is spent inferring types.

Another application is the use of this algorithm in general type inference for object-oriented languages. This is a harder problem than the one we are concerned with here, as the aim is to infer all types, including those of methods. There exists a vast literature on the subject, going back at least to Suzuki's paper on type inference for Smalltalk [25]. The key advance was the framework of Palsberg and Schwartzbach [20], on which most later works are based. That framework uses intraprocedural type inference, the problem considered here, as a subroutine. Consequently, an improvement to that simpler problem will also benefit more general type inference.

A third application is in language design. Popular languages like Visual Basic 9 allow a very limited form of type inference for local variables, but only by inferring the type of the initialising expression. A truly efficient algorithm for the problem addressed here would make it possible to relax that restriction, giving the tightest possible type if one exists, and clear error messages when the type is ambiguous.

Contributions We shall present a novel algorithm for local type inference, which is based on the following observation. Write $t_1 \leq t_2$ to indicate that t_1 is a subtype of t_2. Statements induce constraints on the type of local variables. In particular, an assignment $v = E$ induces the constraint

$$[e] \leq [v]$$

where $[e]$ is the type of the expression e and $[v]$ is the type of the local variable v. In words, assignments induce lowerbounds on the types of variables. All other uses induce upperbounds of the form

$$[v] \leq t$$

for local variable v and some fixed type t. Therefore, to find minimal types for variables, it suffices to first process only assignments, and to find a minimal solution for those. Then, in a second stage, the algorithm checks whether the minimal solution satisfies all the other constraints. Note that *if a valid typing exists*, the minimal solution found in the first stage is such a typing.

The above observation opens the door towards a much simpler algorithm than those that have been considered before. Apart from being simpler to implement, it is also vastly more efficient, dealing very well with common cases. For example, when we substitute our new algorithm for the one of [8], we see a 92-fold improvement in execution time of *abc* processing its own bytecode. On other benchmarks the gain is even greater, up to a factor of 575. Not only is the new algorithm faster in practice, it also guarantees a tightest possible result, whereas the algorithm of [8] does not.

The contributions of this paper are:

- a novel, fast algorithm for local type inference;
- a proof of its soundness and optimality;
- a careful discussion of implementation decisions;
- extensive experiments demonstrating its performance.

Overview The structure of this paper is as follows. First, in Section 2, we discuss the algorithm in abstract form, and we prove its correctness. The proof that the least fixpoint is a sound solution of the constraints induced by assignment statements is of particular interest. Next, in Section 3, we discuss a number of implementation decisions, and we report performance experiments for type inference in Section 4, using the type hierarchy employed in Java bytecode. That hierarchy is different from the Java source type hierarchy in the way primitive types are treated, and this issue is investigated in Section 5. We then proceed to present a further experimental evaluation of such source type inference in Section 6. As we have already mentioned, there exists a vast body of literature on type inference and its variations, and we review the most pertinent previous works in Section 7. We conclude in Section 8, and we point out opportunities for further work.

2. TYPE INFERENCE ALGORITHM

The key idea of the inference algorithm is to proceed in two phases. In the first phase, we only consider assignments where the left-hand side is a local variable, and we compute a minimal type for each local variable by a simple fixpoint iteration. The second phase then only consists of checking the solution.

We first present the algorithm making the assumption that types form a lattice. That assumption is not satisfied for types in Java, so we show how to take the partial order of Java type conversions and construct a lattice of typings. That construction in terms of so-called 'upwards-closed sets' (which is standard) shows the algorithm is correct, but it would be expensive to implement in practice. We go on, therefore, to consider the representation of upwards-closed sets by small sets of representative elements.

2.1 Lattice algorithm

Let $(T, \leq)$ be the lattice of types. For now we shall not define the notion of types further, leaving a more detailed discussion till we consider Java types. A sample type lattice is shown below:

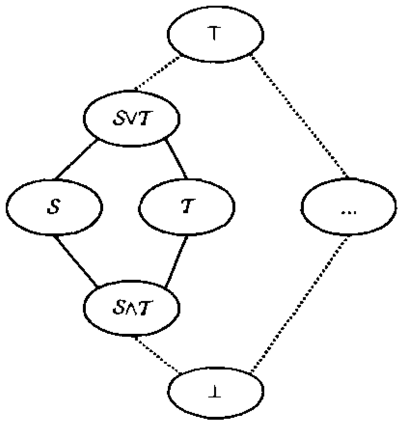


Figure 1. A type lattice

A *typing* $\sigma : V \to T$ is a finite map from variables to types. The set of all typings is itself a lattice, with the pointwise order, given by

$$\sigma_1 \leq \sigma_2 \quad \equiv \quad \forall v : \sigma_1(v) \leq \sigma_2(v)$$

The *type evaluation mapping* **eval** : $((V \to T) \times E) \to T$ evaluates an expression with a given typing, to yield the type of the whole expression. We require that the type system of the programming language is such that **eval** is monotonic:

$$\sigma_1 \leq \sigma_2 \quad \text{implies} \quad \mathbf{eval}(\sigma_1, e) \leq \mathbf{eval}(\sigma_2, e) \qquad (1)$$

Again, we do not specify **eval** further at this point, but we shall discuss it in more detail in the next subsection, when we relate it to the Java type system.

A typing σ is said to be *valid* for an assignment instruction a of the form $v := e$ whenever

$$\mathbf{eval}(\sigma, e) \quad \leq \quad \sigma(v)$$

A typing σ is said to be *assignment-valid* if it is valid for all assignment instructions a in the program.

A *use* of a variable v is a pair (v, t) which models the situation where v is used in a position where a variable of type t is expected. A typing σ is said to be *valid* for a use (v, t) whenever

$$\sigma(v) \quad \leq \quad t$$

A typing σ is said to be *use-valid* if it is valid for all uses in the program.

A typing σ is *valid* if it is both assignment-valid and use-valid. Our aim is to construct a smallest valid typing. We shall do that by constructing a smallest assignment-valid typing π, and then checking that that π is also use-valid. If it is, then π is the smallest valid typing. Conversely, suppose that π' is a smallest valid typing. Then $\pi' \leq \pi$ and (because π is the smallest assignment-valid typing) $\pi \leq \pi'$.

Least fixpoint To compute the smallest assignment-valid typing, define

$$f(\sigma)(v) \quad = \quad \bigvee\{\, \mathbf{eval}(\sigma, e) \mid (v := e) \in P\,\} \qquad (2)$$

In words, we take the least upperbound of $\mathbf{eval}(\sigma, e)$ over all assignments $v := e$ in the program.

We claim that σ is a prefix point of f if and only if σ is assignment valid. The proof is a simple calculation:

$$\begin{array}{ll} & f(\sigma) \leq \sigma \\ \equiv & \quad \{\text{pointwise order on typings (1)}\} \\ & \forall v : f(\sigma)(v) \leq \sigma(v) \\ \equiv & \quad \{\text{definition of } f \text{ (2)}\} \\ & \forall v : \bigvee\{\, \mathbf{eval}(\sigma, e) \mid (v := e) \in P\,\} \;\leq\; \sigma(v) \\ \equiv & \quad \{\text{least upperbound}\} \\ & \forall v : \forall (v := e) \in P : \mathbf{eval}(\sigma, e) \leq \sigma(v) \\ \equiv & \quad \{\text{since every variable is assigned}\} \\ & \forall (v := e) \in P : \mathbf{eval}(\sigma, e) \leq \sigma(v) \end{array}$$

The smallest assignment-valid typing exists by virtue of the fact that f is monotonic, and so it has a least fixpoint by the Knaster-Tarski theorem [14].

In conclusion, we can compute the smallest assignment-valid typing by computing a least fixpoint of f. In doing so, it would obviously be beneficial to track dependencies between variables, and keep a worklist of assignments that may need to be revisited upon each iteration. We shall discuss those and related issues later in Section 3. It is worthwhile to note, however, how at this abstract level our type algorithm is disarmingly simple.

2.2 Completing the partial order of typings

Write $t_1 <: t_2$ to indicate that a Java type t_1 can be converted to Java type t_2. A full specification of this partial order can be found in the Java Language Specification [9]. The above algorithm is cute, but at first sight it may appear useless in the context of Java, because the partial order $<:$ does not have least upperbounds. One reason for the absence of least upperbounds is multiple inheritance via interfaces in Java: two classes A and B may implement the same two interfaces I and J.

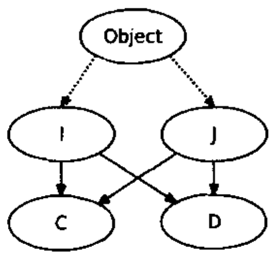


Figure 2. Partial type hierarchy with interfaces

Because there are no least upperbounds for Java types, that are no least upperbounds for typings either. Now one might think a suitable solution is to work with sets of types (as in e.g. [20]) but that would defeat the purpose of our inference algorithm: we want to find a typing that is correct according to the rules of Java. To illustrate, consider the situation above with class A and B that may implement the same two interfaces I and J, and take the program fragment

$$x = \mathbf{new}\, A(); y = \mathbf{new}\, B(); x = y; y = x;$$

Working with sets, the conclusion would be that both x and y are assigned type $\{I, J\}$. In terms of Java typings, we would however have to choose either type I for both x and y, or type J, but I for x and J for y is not allowed. We need to track such dependencies during type inference.

For that reason we shall work with sets of typings, or more precisely upward-closed sets of typings. A set of typings Σ is said to be *upward-closed* when

$$\sigma \in \Sigma \wedge \sigma \leq \sigma' \quad \text{implies} \quad \sigma' \in \Sigma$$

Upward-closed sets of typings are ordered by

$$\Sigma \leq \Sigma' \quad \equiv \quad \Sigma' \subseteq \Sigma \qquad (3)$$

The *minimal* elements of a set are those that have no predecessors. Formally, we have $\sigma \in \mathbf{mnl}(\Sigma)$ when for all σ', the following equivalence holds:

$$\sigma' \in \Sigma \wedge \sigma' \leq \sigma \quad \equiv \quad \sigma' = \sigma \qquad (4)$$

It is easy to check that for upwards-closed Σ and Σ', we have

$$\Sigma \subseteq \Sigma' \quad \equiv \quad \mathbf{mnl}(\Sigma) \subseteq \Sigma' \qquad (5)$$

In words, to check that Σ is included in Σ', we only need to consider the minimal elements of Σ, as by virtue of upward-closure, all the other elements of Σ are then also in Σ'.

Our aim is now to define a generalisation of the mapping f in the previous section (2), which we used to compute the minimal assignment-valid typing. Instead, we shall be computing a least set of minimal assignment-valid typings. That set will be *least* in the $\leq$ order, so it is *greatest* in the $\subseteq$ order, and therefore all assignment-valid typings will be represented in the result.

For brevity, define the following predicate on pairs (σ, σ') of typings:

$$step(\sigma, \sigma') \quad = \quad \forall (v := e) \in P : \mathbf{eval}(\sigma, e) \leq \sigma'(v)$$

It is easy to see that $step(\sigma, \sigma)$ is a restatement of assignment-validity of σ. Now define a mapping F on upward-closed sets of typings as follows:

$$F(\Sigma) \quad = \quad \{\, \sigma' \mid \exists \sigma \in \Sigma : step(\sigma, \sigma') \wedge \sigma \leq \sigma' \,\} \qquad (6)$$

Note that the result is indeed upward-closed. It is worthwhile to compare this definition of F to that of the mapping f in the previous section: it is a natural generalisation for the situation where least upperbounds need not exist.

We now claim that to compute the least (in $\leq$) set of minimal assignment-valid typings, all we need to do is to take the least fixpoint of F; the minimal elements of that least fixpoint are the desired typings. To prove that, we reason as follows:

$$\begin{array}{ll}
 & F(\Sigma) \leq \Sigma \\
\equiv & \quad \{\text{definition of} \leq (3)\} \\
 & \Sigma \subseteq F(\Sigma) \\
\equiv & \quad \{\text{inclusion of upward-closed sets } (5)\} \\
 & \mathbf{mnl}(\Sigma) \subseteq F(\Sigma) \\
\equiv & \quad \{\text{definition of } F\ (6)\} \\
 & \forall \sigma' \in \mathbf{mnl}(\Sigma) : \exists \sigma \in \Sigma : step(\sigma, \sigma') \wedge \sigma \leq \sigma' \\
\equiv & \quad \{\text{definition of minimal } (4)\} \\
 & \forall \sigma' \in \mathbf{mnl}(\Sigma) : step(\sigma', \sigma')
\end{array}$$

We conclude that a simple generalisation of our original algorithm suffices to find the set of all assignment-valid typings.

2.3 Representing upwards-closed sets

In practice, representing each upwards-closed set of typings explicitly is prohibitively expensive — consider the fact that the computation of a least fixpoint will start with the bottom upwards-closed set of typings, which by definition contains all possible typings.

A seemingly obvious solution is to represent each upwards-closed set only by its minimal elements. While that is certainly an improvement over keeping all elements, we have found that the requirement that the results be minimal at every step imposes an unduly large penalty in terms of comparisons. We want to keep the sets small throughout the algorithm execution, certainly, but there is no harm in having a few non-minimal elements.

To make this intuition precise, we define a new preorder (a reflexive and transitive relation) on arbitrary (not necessarily upwards-closed) sets of typings:

$$\Sigma \preceq \Sigma' \quad \equiv \quad \mathbf{up}(\Sigma) \leq \mathbf{up}(\Sigma') \qquad (7)$$

where $\mathbf{up}(\Delta) = \{\, \sigma' \mid \exists \sigma \in \Delta : \sigma \leq \sigma' \,\}$. That is, $\preceq$ mimics our partial order $\leq$ on upwards-closed sets of typings, by just working with sets of representative elements. We can in fact implement the test for $\preceq$ without the expensive computation of upwards-closed sets, for we have

$$\Sigma \preceq \Sigma' \quad \equiv \quad \forall \sigma' \in \Sigma' : \exists \sigma \in \Sigma : \sigma \leq \sigma' \qquad (8)$$

In words, for every typing in Σ', there exists a smaller representative in Σ (*cf.* Figure 3). This preorder is very common in programming language semantics and program analysis, in particular since it is the order in the Smyth powerdomain [23].

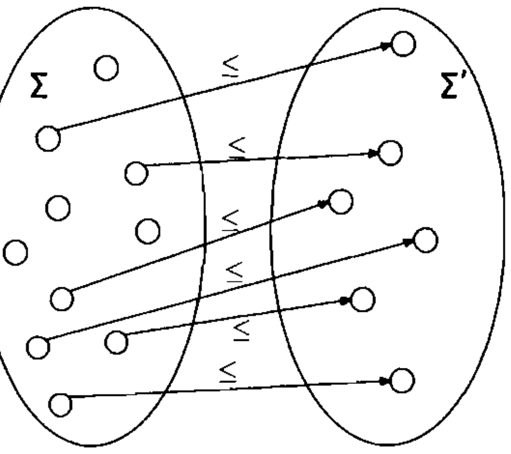


Figure 3. Order on sets of typings: $\Sigma \preceq \Sigma'$

Now say that two sets Σ and Σ' are *equivalent* if each is at least as large as the other:

$$\Sigma \simeq \Sigma' \quad \equiv \quad \Sigma \preceq \Sigma' \wedge \Sigma' \preceq \Sigma \qquad (9)$$

Equivalent sets represent the same upwards-closed set (by equivalence (7)), so via (8) we now have an effective test for equality of upwards-closed sets, still just working with representative elements. As a special case, note that any set is equivalent to its minimal elements:

$$\Sigma \quad \simeq \quad \mathbf{mnl}(\Sigma) \qquad (10)$$

Our aim, therefore, is to implement the above abstract algorithm by keeping a set of typings Σ' that is equivalent to the set Σ the abstract algorithm would have computed in the same step.

First we note that by monotonicity of F on $\leq$, we have that F is monotonic on $\preceq$ also, and therefore F preserves equivalence of sets of typings. Furthermore,

$$\begin{array}{ll}
 & F(\Sigma) \\
= & \quad \{\text{definition of } F\ (6)\} \\
 & \bigcup_{\sigma \in \Sigma} \{\sigma' \mid step(\sigma, \sigma') \wedge \sigma \leq \sigma'\}
\end{array}$$

$$\simeq \quad \{\text{union preserves equivalence, (10)}\}$$
$$\bigcup_{\sigma\in\Sigma}\mathbf{mnl}(\{\sigma' \mid \mathit{step}(\sigma,\sigma') \wedge \sigma \le \sigma'\})$$

In words, this shows that we can implement F by selecting minimal typings after doing one pass over all assignment statements with a given typing, making any updates to the typing as necessary. It remains to show how one could implement the operation

$$\mathbf{next}(\sigma) \quad = \quad \mathbf{mnl}(\{\sigma' \mid \mathit{step}(\sigma,\sigma') \wedge \sigma \le \sigma'\})$$

Clearly σ' should map each variable to a minimal type satisfying the indicated predicate. Therefore, define a new function **lca** on sets of Java types, such that $\mathbf{lca}(S)$ contains precisely the least common ancestors (*i.e.* supertypes) of the types in S. To illustrate, with the hierarchy displayed in Figure 1, we have $\mathbf{lca}(\{C,D\}) = \{I,J\}$. With the definition of **lca** in hand, obviously we have

$$\mathbf{next}(\sigma) = \{\sigma' \mid \forall v : \sigma'(v) \in \mathbf{lca}(\{\sigma(e) \mid (v := e) \in P\})\}$$

In summary, we have shown that when $\Sigma \simeq \Sigma'$,

$$F(\Sigma) \quad \simeq \quad F'(\Sigma') \tag{11}$$

where $F'(\Sigma') = \{\sigma' \mid \exists\sigma \in \Sigma' : \sigma' \in \mathbf{next}(\sigma)\}$. Writing $\mathbf{lfp}(\sqsubseteq, \phi)$ for the operator that returns a least fixpoint of ϕ in preorder $\sqsubseteq$, we conclude that

$$\mathbf{mnl}(\mathbf{lfp}(\le, F)) \quad = \quad \mathbf{mnl}(\mathbf{lfp}(\preceq, F')) \tag{12}$$

This shows how to implement our abstract algorithm on sets of representative elements, avoiding both the expensive construction of upwards-closed sets of typings, and also avoiding the need to reduce to minimal elements every time the union operator is applied.

2.4 Second Phase

We have now seen how to infer a minimal set of *assignment-valid* typings Σ for a method. However, we are interested in inferring *valid* types, *i.e.* they should be both assignment-valid and use-valid.

Suppose the method is typable (*i.e.* there exists some valid typing), and let π be a minimal valid typing. By definition π is assignment-valid, and so $\sigma \le \pi$ for some $\sigma \in \Sigma$, since all minimal assignment-valid typings are in Σ. But since variable uses induce *upper bounds* on types, if π is use-valid then any smaller typing is also use-valid, and so σ is valid. By minimality of π, $\pi \le \sigma$ which implies $\pi = \sigma$, and we have shown that all minimal valid typings are contained in Σ.

Therefore, the second phase of our algorithm goes through Σ, discarding each element that is not also *use-valid*. If after this pruning Σ contains only one type assignment, then this is the optimal *valid* typing. If Σ has several elements, then all of them are minimal, and we pick one non-deterministically. If Σ is empty, then there exists no valid typing, and the algorithm fails.

3. IMPLEMENTATION

The previous section presented our type inference algorithm at a high level of abstraction, and provided proofs of both soundness and optimality — that is, we know that any inferred types follow the typing rules and, moreover, are as tight as possible.

In practice, some care needs to be taken to ensure an implementation remains efficient. In particular, as is usual in such cases, our fixpoint iteration makes use of a worklist, so that iterations only revisit those statements that may influence the result.

3.1 Fixpoint Iteration with a Worklist

As in Section 2, for simplicity we will first assume that the type hierarchy is a lattice (that is, we do not account for multiple inheritance), and later generalise this to the full language.

As shown in Section 2.1, in the simpler case we need only consider a single typing that is repeatedly refined, rather than sets of typings. The data structure we use for the worklist is a *queued set* — it contains no duplicates, and elements can be taken out in the order in which they were put in. The algorithm would be equally correct with other representations of the worklist, like sets (with non-deterministic order of retrieval) or lists (which may contain duplicates), but a queued set leads to less work overall, as variables tend to be assigned textually before they are used (there may be exceptions, due to jumps).

We will also need information about dependencies between variables. Informally, the type of a variable v_1 depends (or, rather, may depend) on the type of v_2 if v_2 occurs on the right-hand side of some assignment to v_1. We construct a map **depends** such that $\mathbf{depends}(v)$ is a set containing all assignments to some local with v on the right-hand side.

Given that, our implementation proceeds as shown in Algorithm 1.

Algorithm 1 Type inference algorithm for a type lattice

```
1  for every local variable v do
2      $\sigma(v) \leftarrow \bot$;
3  worklist ← set of all assignments to local variables;
4  while worklist is not empty do
5      $(v := e) \leftarrow \mathrm{head}(\mathit{worklist})$;
6      $\mathit{worklist} \leftarrow \mathrm{tail}(\mathit{worklist})$;
7      $t \leftarrow \mathbf{lca}(\sigma(v), \mathbf{eval}(\sigma, e))$;
8      if $t \ne \sigma(v)$ then
9          $\sigma(v) \leftarrow t$;
10         $\mathit{worklist} \leftarrow \mathit{worklist} +\!\!+\ \mathbf{depends}(v)$;
11 return $\sigma$;
```

In words, we start off with the bottom typing (Lines 1 and 2). Next, we iterate over the assignments to local variables. For each assignment $v := e$, the typing is updated (on

Line 7) to the least common ancestor (in lattice terms, the least upperbound) of the type $\sigma(v)$ and the type of the right-hand side under σ, that is **eval**(σ, e). Should this induce a change in σ (Line 8), all the assignments that depend on v are queued for consideration in a later iteration (Lines 9 and 10). It is evident that the above computes the same result as the abstract algorithm in Section 2.1, as taking the least common supertype of a set of types is the same as taking the pairwise least common supertype of elements of that set.

To extend this algorithm to the general case, we need to consider sets of typings, each with an associated worklist, and the fixpoint we are computing will be such a set of typings. Write **worklist** (σ) for the worklist of the typing σ. Note that we will represent upwards-closed sets of typings by their minimal elements, as shown in Section 2.3.

Algorithm 2 General type inference algorithm

```
1  for every local variable v do
2      σ(v) ← ⊥;
3  Σ ← {σ};
4  worklist(σ) ← set of all assignments to local
   variables;
5  while for some σ ∈ Σ, worklist(σ) ≠ ∅ do
6      Pick σ ∈ Σ where worklist(σ) ≠ ∅;
7      Σ ← Σ \ {σ};
8      (v := e) ← head(worklist(σ));
9      worklist(σ) ← tail(worklist(σ));
10     for each t in lca(σ(v), eval(σ, e)) do
11         if t = σ(v) then
12             Σ ← Σ ∪ {σ};
13         else
14             σ' ← σ[v ↦ t];
15             worklist(σ') ←
               worklist(σ) ++ depends(v);
16             Σ ← Σ ∪ {σ'};
17 return Σ;
```

We proceed as shown in Algorithm 2, which it is worthwhile to examine in some detail. Again, we start with the bottom typing (Lines 1 and 2), and put that in our set of candidate typings (Line 3). The worklist for Σ initially consists of all assignment statements (Line 4). The iteration now continues as long as there exists some non-empty worklist. Let us now look at the way iteration steps are performed a bit more closely. We pick a typing σ that has a non-empty worklist (Lines 6 and 7), and an assignment $v := e$ from that worklist (Lines 8 and 9). Then for each type t in the set of least common ancestors of $\sigma(v)$ and **eval**(σ, e), we check whether σ needs to be updated (Line 11). If not, we just add σ to our current set of candidate typings (Line 12). On the other hand, if an update is required, we create a new version σ' of σ that maps v to t (Line 14), and expand the worklist of σ' accordingly (Line 15). Finally, we add σ' to the set of candidate typings (Line 16).

```
1  class CA { void f() {} }
2  class CB { void g() {} }
3  ...
4  void method() {
5    <untyped> x;
6    if( ... ) {
7      x = new CA(); x.f();
8    } else {
9      x = new CB(); x.g();
10   }
11   x.toString()
12 }
```

Figure 4. A method with no valid type for x

From this description, it becomes evident that our algorithm has a potential source of severe inefficiency, namely the iteration in Lines 10 to 16, which could multiply the size of the set of candidates each time it is executed. As we shall see shortly through a series of experiments, that does not happen in practice because it is very rare for **lca** to return non-singleton results.

3.2 Arrays

So far we have only considered assignment statements of the form $v := e$ where v is a local variable. There is also another case that must be handled in order to generate assignment-valid typings for Java: assignments to array references. In Jimple all array references take the form $v[i]$ where v is a local variable, so we need to consider assignments statements of the form $v[i] := e$. The required modifications to Algorithms 2 and 3 are minor: in the case of such assignments we use $t \leftarrow$ **lca**$(\sigma(v),$ **eval**$(\sigma, e)\,[\,])$. Notice the [] notation, indicating that we take the **lca** with the array type whose elements are of the type of the expression e. If e has an array type already then we are taking the **lca** with a multidimensional array type.

3.3 Type Inference for Arbitrary Bytecode

The above algorithm infers a set of minimal typings, which (by the proof we presented earlier in Section 2) are assignment-valid. There is no guarantee, however, that the checking phase (Section 2.4) is then going to succeed, even for Java bytecode that is verifiable.

Let us consider the reasons why no valid typing might exist. The problem stems from the fact that the bytecode verifier does a simple flow analysis to estimate the type of stack locations at each program point; in particular, each stack location can have different types at different points, while we are concerned with inferring a *single* type for each variable that is valid throughout the method. Consider the code snippet in Figure 4 (this example is due to Gagnon *et al.* [8]). As far as the bytecode verifier is concerned, on line 7 x has type *CA*, and on line 9 it has type *CB*, while outside the if statement it has type *Object*. However, none of these types work throughout the whole method.

To deal with such a case, we transform the method body into an equivalent form for which a valid typing exists.

Of course such transformations are undesirable, and so we only apply them if the type inference algorithm finds no valid typing. Following [8], we have two transformation stages after the first type inference stage. The first is a particular variable-splitting transformation at object creation sites that allows inferring types in some previously problematic cases at the cost of introducing more Jimple variables. Indeed, our experiments confirm the conjecture first voiced by Gagnon *et al.* stating that in the vast majority of practical cases this stage is sufficient to infer types; more details are given in Section 6.

Still, there are certain cases (*e.g.* Figure 4) which remain untypable. The third stage, therefore, starts with assignment-valid typings and introduces casts to make them use-valid. In the presence of several possibilities, the one that requires fewest casts is deemed preferable.

Now, if it is the case that an assignment-valid typing exists, then the above two additional transformation steps are guaranteed to find some validly typed solution — at least validly typed according to the type conversions allowed by Java bytecode.

4. EXPERIMENTAL EVALUATION

Algorithm 2 for local type inference was implemented as part of the bytecode to Jimple pass of the Soot optimisation and decompilation framework [26]. For this first set of experiments we used the augmented value set hierarchy introduced in Section 5, as opposed to the Java source type hierarchy. In particular implicit conversions are allowed between all integer types (`boolean`, `int`, `byte`, `short` and `char`). This is in fact not the case in the original Soot implementation of [8], and in Section 5 we show how to fix that.

In the experiments, we chose a wide variety of benchmarks, totalling over 295K methods. A list of all our benchmarks is shown in Figure 5.

For each experiment, the bytecode to Jimple feature of Soot was executed on each bytecode class file in the benchmark. Table 1 presents the total time spent inferring types under our implementation of Algorithm 2 and, for comparison, under the original Soot algorithm of Gagnon *et al.* The separate integer typing stage of the original Soot algorithm was carefully disabled. Our implementation is, in fact, doing slightly more work (in finding an assignment-valid integer typing under the augmented value set hierarchy discussed later in Section 5) than necessary to provide a precise comparison. Each experiment was run on the same quad-core Intel Xeon 3.2GHz machine with 4GB RAM running Linux 2.6.8 SMP. Each benchmark was tested 5 times independently, and the middle 3 results (selected independently between the two algorithms compared) were averaged. From these two times we determine the relative improvement. Also recorded are the number of methods tested in each benchmark and the number of methods for which Algorithm 2 finds a tighter typing (it never gives a weaker typing). For each benchmark we present the mean number of minimal candidate typings generated by Algorithm 2, which is representative of the extent of multiple-inheritance. Finally we show the the number of methods typed at each of the three stages of code transformations:

Stage 1 A valid typing exists for the original method so no transformation is required.

Stage 2 A variable-splitting transformation is used at object creation sites.

Stage 3 The least number of safe casts are inserted where required.

Note that the original Soot algorithm also uses the three-stage technique, and the same stage is always used in both algorithms.

Algorithm 2 is typically around 6 times faster in the benchmarks tested, however two cases stand out where a dramatic improvement is found. On closer examination we notice that both `abc-complete.jar` and `havoc.jar` contain several huge (>9000 Jimple statements) methods. It is interesting, therefore, to measure the performance of the respective algorithms as a function of the size of the method bodies being processed. These results are plotted as Figure 6. Note that the vertical axis has a cube-root scale — as noted in [8], the asymptotic complexity of Soot's type inference algorithm is cubic, and the plot shows that this is indeed attained in practice.

For clarity, the data points corresponding to Algorithm 2 are shown on their own in Figure 7; the scale here is linear. It is easy to see that the overwhelming trend in the common case is linear; there are a few outliers (which are still low compared to the other algorithm), and they usually correspond to cases where multiple inheritance induced multiple candidate typings.

Table 1 validates our earlier remark that multiple inheritance is extremely rare in real-world programs: the seventh column lists the mean number of minimal typings for each method, and in the vast majority of cases this is 1. Note that typically it is non-javac sources (`kawa`, `scala`, `cso`) that show an abnormally high value, and indeed this seems correlated with a comparatively lower relative improvement.

The numbers also show that almost all methods can be typed by applying only stage 1 of the algorithm, and the majority of methods requiring stage 2 stem from `scala`. In total, in our benchmarks (which were chosen to present challenges to a type inference algorithm) only 0.1% of methods required the second stage, and 0.02% the third.

rt is the Sun Java 1.5 runtime library. The main interest of this benchmark is its size (108K methods), and the fact that it exercises many features of the Java language.

tools is the Sun JDK 1.5 tools libray including *javac*. Again we chose this benchmark because it is an interesting piece of Java, albeit of modest size (14K methods.)

abc-complete is version 1.2.1 of the AspectBench Compiler for the AspectJ programming language including Soot and Polyglot. This is interesting as a benehmark because it contains many large, generated methods.

jython is version 2.2.1 of Jython, a Python implementation written in Java. This is chosen as a mid-sized example of typical Java code.

groovy is version 1.5.4 of the compiler for the Groovy programming language. Again this is written in Java and provides another benchmark containing typical Java code.

gant is version 1.1.1 of the Gant build system, similar to Ant but compiled to bytecode by Groovy instead of *javac*. This is an important experi ment because the algorithm is designed to handle all valid bytecode, not just bytecode generated from Java source.

kawa is version 1.9.1 of the Kawa compiler for the Scheme programming language. Here part of the jar is bootstrapped, again giving bytecode sequences that would not normally occur as the output of *javac*.

scala is version 2.7.0 of the Scala compiler and runtime library, both of which are written in Scala, and compiled by the Scala compiler (again, instead of *javac*).

cso is a concurrency library, loosely inspired by the CSP calculus, written by Bernard Sufrin in Seala. This benchmark is also eompiled by Scala instead of *javac*.

jigsaw is version 2.2.6 of the W3C's Jigsaw web server implementation, and this is included as a typieal web application written in Java.

jedit is version 1.4pre13 of the jEdit text editor, an example of an interactive applieation written in Java.

bluej is version 2.2.1 of the BlueJ IDE for the Java programming language, again chosen as an interactive application.

java3d is version 1.5.1 of the Java 3D API. As we shall see later in this paper, (numerical) primitive types ean pose a challenge for type inference algorithms on Java bytecode, and this is a potential example of that phenomenon.

jgf is version 2.0 of the Java Grande Forum Sequential Benchmark Suite, again chosen for its use of primitive type operations.

havoc is a contrived example of Java bytecode which takes unusually long for the JVM to verify [6], in fact it was designed to be a denial-of-service attack on the Java bytecode verifier.

Figure 5. Benchmark descriptions

Benchmark	# Methods	Old Time (s)	New Time (s)	Improvement	# Tighter	Mean # Cndts.	# Stg. 1	# Stg. 2	# Stg. 3
`rt`	107792	84.48	10.77	7.84x	39	1.00032	107681	77	34
`tools`	14180	13.07	2.37	5.52x	5	1.00014	14160	17	3
`abc-complete`	33866	480.28	4.37	109.88x	52	1.00027	33865	1	0
`jython`	9192	6.67	1.25	5.35x	0	1.00000	9187	5	0
`groovy`	13799	10.12	1.92	5.27x	7	1.00087	13778	4	17
`gant`	707	1.75	0.44	4.01x	0	1.00000	702	3	2
`kawa`	9226	7.70	1.58	4.88x	25	1.00618	9195	31	0
`scala`	65161	37.66	5.36	7.03x	117	1.00453	64865	296	0
`cso`	2395	2.20	0.51	4.34x	6	1.00167	2392	3	0
`jigsaw`	13577	11.50	1.84	6.24x	1	1.00007	13571	6	0
`jedit`	5980	5.74	1.09	5.26x	12	1.00318	5969	0	11
`bluej`	5690	5.37	0.98	5.48x	0	1.00053	5690	0	0
`java3d`	13453	17.86	2.46	7.26x	5	1.00037	13453	0	0
`jgf`	557	3.61	0.48	7.59x	0	1.00000	557	0	0
`havoc`	23	198.53	0.46	428.17x	0	1.00000	23	0	0
Total	295598	886.53	35.87	24.72x	269		295088	443	67

Table 1. Performance comparison between both algorithms under the augmented value set hierarchy

5. INFERRING JAVA SOURCE TYPES

So far, we have operated on the assumption that the type system used is that of Java bytecode. If the purpose is just to have a typed intermediate representation of bytecode, with the aim of analysis or optimisation, that is just the right choice. However, the framework in which we are conducting our experiments (Soot) has another purpose, namely decompilation of bytecode to Java source. For that application, it is desirable that the inferred types are correct with respect to the type system of the Java source language.

In respect to integers, the legal widening conversions between primitive types in Java source (which are identical to those in Jimple source) are rather different to Java bytecode (cf. Figure 8). The narrowing conversions allowed in Jimple source are identical to Java source, with one essential exception: Jimple allows casts between **boolean** and any integer type, while Java does not allow **boolean** to be cast to or from any other type. In bytecode, all integer types of less than 4

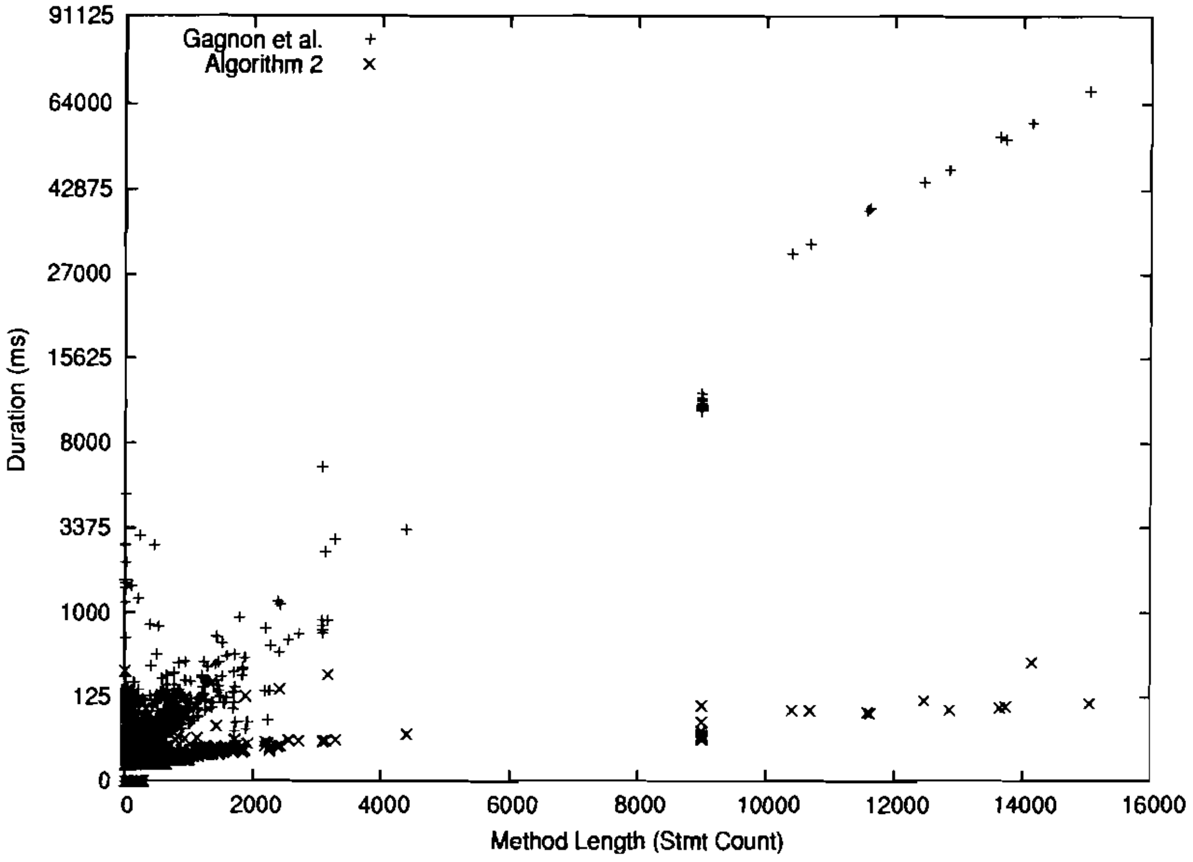


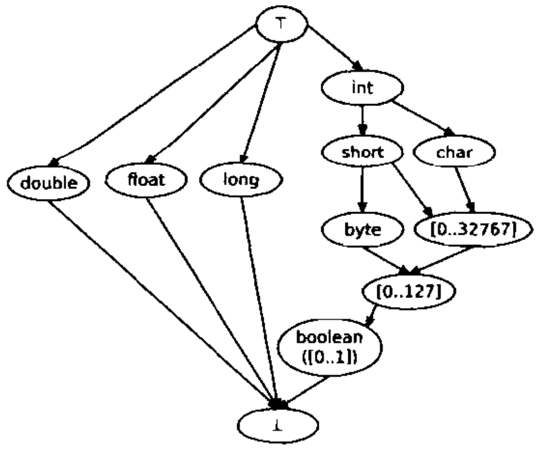
Figure 6. Comparison of the two algorithms: runtime against method size; cube-root plot.

bytes are represented as **int**, and the only differencc is the range of values — they can be used interchangeably. The larger primitive types are separate, but there are instructions for converting between them (which at the Jimple and at the Java level level look like casts).

The reason this causes problems is that it is possible to have verifiable bytecode without an assignment-valid typing with respect to the source type hierarchy. Consider the following sequence of statements, which is perfectly valid in bytecode: x = (a < b); x = 2;. The first statement assigns a **boolean** to x, the second a value that cannot be a **boolean**, and since neither is a supertype of the other there is no typing that makes both assignments work.

Integer type inference is further complicated by the fact that it is not clear how to define the **eval** function with this hierarchy, as in x = 0;, the right-hand side could have the types **boolean**, **byte** or **char** (or, of course, their supertypes). Thus, the fact that Jimple uses the source type hierarchy (for the sake of decompilation) works against us here — it would be much more convenient if we could simply infer **int** whenever such an integer type is expected!

To address this issue, we consider an augmented type hierarchy (called the *value set hierarchy*) for the integer types, as shown in Figure 9. Essentially, we introduce three vir-



$$\begin{tikzcd}
 & & \top \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & & & int \arrow[d] \arrow[dr] & \\
double \arrow[ddddrr] & float \arrow[ddddr] & long \arrow[dddd] & short \arrow[d] \arrow[dr] & char \arrow[d] \\
 & & & byte \arrow[d] & {[0..32767]} \arrow[dl] \\
 & & & {[0..127]} \arrow[d] & \\
 & & & boolean\ ([0..1]) \arrow[dl] & \\
 & & \bot & &
\end{tikzcd}$$

Figure 9. The augmented value set hierarchy

tual types (called *value set types*) based on the value ranges of integer constants: [0..1], [0..127] and [0..32676]. These don't correspond to real source types, but allow us to defer the decision of whether, for example, the integer constant 0 is a **boolean**, **byte** or **char** temporarily. We also make the observation that the value set type [0..1] actually coincides with **boolean**, so we combine the two, ensuring that

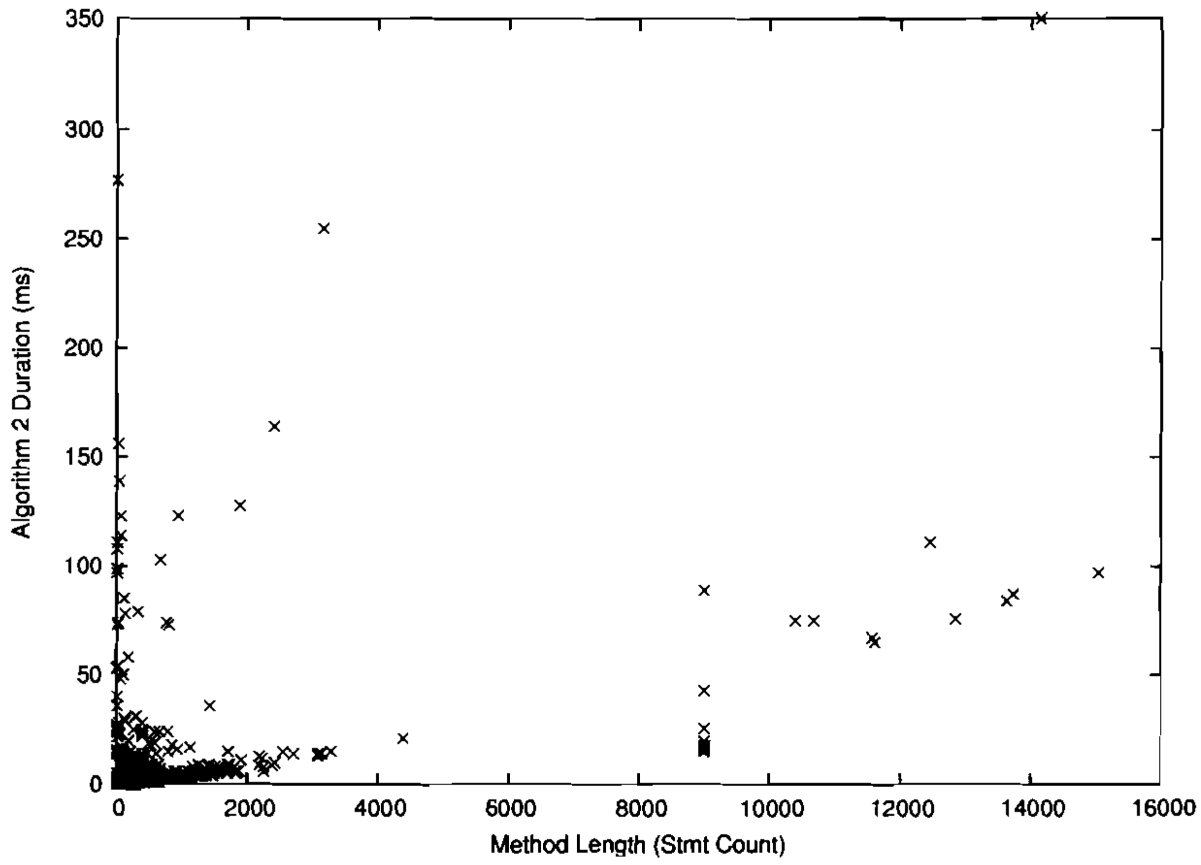


Figure 7. Our algorithm: runtime against method size.

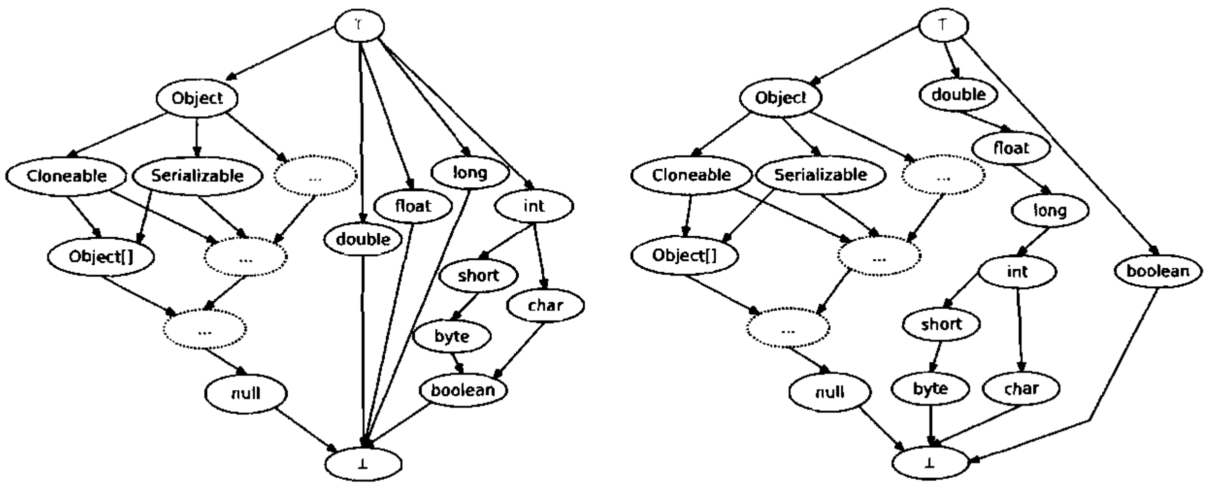


Figure 8. The type hierarchy in Java bytecode and Java source

booleans are assignable to other integer types. With the value set hierarchy, any value that can be assigned safely to a variable of type t can also always be assigned to any ancestor type of t.

Thus, we can now give a procedure for inferring integer types: Generate the least typings that are assignment-valid under the value set hierarchy (such typings need not be assignment-valid in Jimple without inserting casts, but any such casts are guaranteed to preserve semantics since they move up in the value set hierarchy). Resultant typings may contain value set types which we must promote to concrete types and, again, insert casts where required under the source type hierarchy. This gives a valid Jimple typing.

Algorithm 3 Type promotion algorithm

```
   Input: A typing $\sigma$ possibly containing value set types
1  for each variable use $(v, t)$ where $\sigma(v) \leq$ int do
2      if $\sigma(v) \nleq t$ then no valid typing exists; fail ;
3      if $\sigma(v)$ is a value set type then
4          $\sigma(v) \leftarrow$ the least $t' \geq \sigma(v)$ such that all
           ancestors of $t'$ are comparable to $t$;
5  for each local $v$ where $\sigma(v)$ is a value set type do
6      switch $\sigma(v)$ do
7          case [0..1]
8              $\sigma(t) \leftarrow$ boolean
9          case [0..127]
10             $\sigma(v) \leftarrow$ byte
11         case [0..32767]
12             $\sigma(v) \leftarrow$ char
13 return $\sigma$;
```

Concretely, the first stage of type inference runs the algorithm described in Section 2 while lumping all small integer types together into **int**. As observed above, we could stop at this point if the purpose of type inference is optimisation, and hence bytecode types suffice. The second stage revisits all variables typed as **int**, and applies the main algorithm in conjunction with the value set hierarchy (treating non-**int** types as fixed). We then proceed to *type promotion*, as shown in Algorithm 3.

We consider in turn each use of an integer-typed variable (lines 1–4); if it is incompatible with the current typing, then no valid typing can exists and the algorithm fails. If the variable has a value set type, it is promoted to the least type such that all ancestors of that type are comparable to the use (and so can be converted, if necessary). For example, suppose we have a variable v typed as `[0..1]`. If we find a use (v, `boolean`) then we type v as `boolean`. If we find a use (v, `int`) then we type v as `[0..127]`, and if we find a use (v, `byte`) then we type v as `byte`. We can note from the augmented hierarchy chosen that this step will never omit a potential concrete type from consideration.

Some variables may still have value set types, so we promote those to a suitable concrete type (lines 5–12) and thus obtain a valid typing. By 'suitable' we mean it can be verified that this step will never introduce invalid assignments or uses, whatever the program or typing.

If the type promotion fails, then we roll back to the initial assignment-valid typing with value set types and proceed to the second stage of integer typing (cf. Algorithm 4). The idea is that each variable currently typed with a value set type actually needs to be typed with a concrete Java type, and so before checking uses we generate a set of candidates consisting of every valid combination of least concrete types. Applying use constraints may introduce casts, so we simply choose the candidate that requires fewest casts. A subtlety lies in the fact that each time we promote a value set type to a concrete type, we must re-apply Algorithm 2 to propagate assignment constraints (although we can restrict it to only integer-typed variables, and set the initial worklist to the dependencies of the promoted variable).

It is clear that the set of candidates can grow exponentially in the number of integer-typed locals. Luckily, the type promotion algorithm (Algorithm 3) suffices in almost all cases. If we were to drop the requirement for inserting the fewest casts, which is the approach taken by Gagnon *et al.* [8] in the second stage of their integer typing algorithm, then this exponential-time algorithm could be reduced to simply applying a single pre-defined promotion to all value set types. A suitable such promotion would be the one used at the end of the type promotion algorithm: `[0..1]` $\rightarrow$ `boolean`, `[0..127]` $\rightarrow$ `byte` and `[0..32767]` $\rightarrow$ `short`.

Algorithm 4 Second stage integer typing

```
   Input: A typing $\sigma$ possibly containing value set types
1  $candidates \leftarrow \{\sigma\}$;
2  while Some $\sigma \in candidates$ uses value set types do
3      Remove $\sigma \in candidates$ using a value set type;
4      Pick $v$ where $\sigma(v)$ is a value set type;
5      switch $\sigma(v)$ do
6          case [0..1]
7              $new \leftarrow \{\sigma[v \mapsto \textbf{boolean}], \sigma[v \mapsto \textbf{byte}],$
               $\sigma[v \mapsto \textbf{char}]\}$;
8          case [0..127]
9              $new \leftarrow \{\sigma[v \mapsto \textbf{byte}], \sigma[v \mapsto \textbf{char}]\}$;
10         case [0..32767]
11             $new \leftarrow \{\sigma[v \mapsto \textbf{char}], \sigma[v \mapsto \textbf{short}]\}$;
12     $new \leftarrow$ the result of Algorithm 2 on $new$;
13     $candidates \leftarrow candidates \cup new$;
14 return $candidates$;
```

Let us examine Algorithm 4 in some detail. We initialise the set of candidates to contain our assignment-valid typing (line 1), and then iterate until each candidate only uses concrete types. While there exists some typing σ giving a value set type to v, we remove it from the set of candidates, and create new typings for each least concrete supertype of v (lines 3–11). Algorithm 2 is run on each typing contained in this way, potentially raising the types of other variables, and the results are added to the candidate set (lines 12-13).

As mentioned above, the resulting set of candidates is then checked against uses, introducing casts as necessary, and the typing with fewest casts is returned. Note that such casts are usually acceptable, since they either move up the value set hierarchy (thus preserving values), or, if the casts are narrowing, then the semantics of the underlying bytecode must have been dubious to begin with.

It is interesting to note the distinct similarities between this second stage algorithm and the handling of multiple inheritance in Algorithm 2. The reason we chose to separate it out was simply the dramatic increase of performance given in almost all cases by the type promotion algorithm. While multiple inheritance in reference types is relatively rare in practice, `[0..1]`-valued integer constants are very common in bytecode — indeed, every conditional jump uses such a constant. The next section shows the performance cost of inferring precise integer types.

6. EXPERIMENTS WITH SOURCE TYPES

We repeated the experiments of Section 4 to determine the effect of integer type inference (under the Java source hierarchy) on the performance of the algorithm. Once again, this part of the algorithm is not strictly necessary if all we want to do is, say, class hierarchy analysis. On the other hand, it is a crucial component of a decompiler. It is interesting, therefore, to determine the exact cost of this additional functionality, so it can be judiciously invoked.

The results of our experiments are displayed in the two parts of Table 2:

- The top part in Table 2 presents the average times spent inferring integer types in our algorithm with type promotion, and the percentage that represents of the total time for type assignments. The next two columns give the same numbers for the algorithm of Gagnon *et al.*. The two next columns show the improvement of our integer type inference method over the one in Soot, and the improvement for the complete type inference process (including integer typing — Section 4 only considered type inference for the bytecode type hierarchy). The final column lists for how many methods the new algorithm found a tighter typing (of course, it can never find a less tight typing, as it is optimal).
- The bottom part of Table 2 gives an indication of the use of different stages of the two type inference frameworks. Both algorithms use two stages to infer integer types, but these two stages are not trivially related.

Let us now examine these numbers in some detail. First, in both the Soot algorithm and the new algorithm, the cost of dealing with integer types is considerable, accounting for 30.5% and 39.9% of the total time spent in type inference. In fact, for some of our benchmarks, the percentage is as high as 47%, so almost half the time of type inference is spent just on getting the integer types right. This underlines the importance of only using the source hierarchy for primitive types when necessary. The "Integer Improvement" column demonstrates that the new method of dealing with integer types performs better than the one in Soot in all cases, despite the fact that it finds a tightest possible typing, whereas Soot's algorithm does not. The latter point is illustrated by the final column, which shows that a small but significant fraction of the methods gets a suboptimal typing in Soot.

In terms of performance, our integer typing stage is 16x faster than the Soot version, but this is mostly due to the (contrived) `havoc` benchmark. The total improvement hovers between 5 and 6 times, with two extremely high ratios that push the overall runtime down 21-fold. The least improved benchmark is `gant`, where we only gain a factor of 2.82.

Moving to the bottom table, it is clear that type promotion (Algorithm 3) is almost always effective in finding the types required, and in fact only the Kawa and Scala benchmarks require the use of Algorithm 4. Similarly the second stage of Soot's integer typing is invoked only for Kawa, albeit less often. It is noteworthy that this concerns bytecode that is not generated by a Java compiler, but instead directly from Scheme.

The reader may wonder whether it would not be possible to forego the type promotion step, and instead always directly use 4. Indeed, in theory that will yield correct results, but in practice that can give an exponential blowup in the number of typings that need to be considered. In fact, we conducted that experiment, and found that only two very small benchmarks `cso` and `jgf` run to completion if type promotion is omitted. We conclude, therefore, that type promotion is the key to efficient yet optimal handling of integer types under the Java source type hierarchy. As mentionned in Section 5, the problem of exponential blowup could be alleviated by relaxing the requirement that we insert as few casts as possible in the case of integers (this is the approach taken by Gagnon *et al.*).

At the beginning of this paper we stressed that the new algorithm is designed to be efficient *for the common case*. It may now appear suspicious that according to the numbers in Table 2, it is *always* better. This is however not the case, it is just that all these benchmarks are whole jars, each consisting of many methods. There are individual methods where our method performs worse than the one in Soot, but that effect is drowned out by the better performance on most other methods.

All these benchmarks, and scripts for reproducing our experiments, can be downloaded from [5].

Benchmark	Soot Time (s)	Soot % total	New Time (s)	New % total	Integer Imprv.	Total Imprv.	# Tighter
rt	22.88	21.31	8.98	45.46	2.55x	5.44x	1061
tools	5.51	29.66	1.04	30.50	5.30x	5.45x	165
abc-complete	89.98	15.78	2.90	39.88	31.03x	78.43x	177
jython	3.59	35.02	1.13	47.50	3.19x	4.32x	45
groovy	3.84	27.50	1.39	41.94	2.77x	4.22x	386
gant	0.59	25.06	0.39	47.30	1.49x	2.82x	70
kawa	3.42	30.74	0.78	33.01	4.40x	4.72x	198
scala	6.28	14.30	2.56	32.30	2.46x	5.55x	190
cso	0.43	16.23	0.31	38.02	1.37x	3.21x	6
jigsaw	3.20	21.77	1.07	36.62	3.00x	5.05x	131
jedit	2.17	27.43	0.56	34.09	3.84x	4.78x	170
bluej	1.12	17.28	0.46	31.88	2.44x	4.51x	78
java3d	5.60	23.89	1.58	39.15	3.54x	5.80x	264
jgf	0.81	18.36	0.24	33.90	3.33x	6.14x	16
havoc	239.55	54.68	0.41	47.03	581.90x	500.47x	0
Total	388.98	30.50	23.79	39.88	16.35x	21.38x	2957

Benchmark	# Type Prom.	# Our Stg. 2	# Soot Stg. 1	# Soot Stg. 2
rt	107792	0	107792	0
tools	14180	0	14180	0
abc-complete	33866	0	33866	0
jython	9192	0	9192	0
groovy	13799	0	13799	0
gant	707	0	707	0
kawa	9202	24	9223	3
scala	65160	1	65161	0
cso	2395	0	2395	0
jigsaw	13577	0	13577	0
jedit	5980	0	5980	0
bluej	5690	0	5690	0
java3d	13453	0	13453	0
jgf	557	0	557	0
havoc	23	0	23	0
Total	295573	25	295595	3

Table 2. Performance comparison for integer typing under the Java source hierarchy hierarchy

7. RELATED WORK

There exists a rich literature on the topic of type inference in object-oriented languages, [2–4, 7, 10–13, 15, 16, 18–21, 24, 25] to name but a few. Almost all of these take the notion of type constraints as their starting point. What sets the present paper apart is the idea to first consider only the constraints that induce a lowerbound on typing, and find a minimal solution for that restricted set of constraints. When that minimal solution is found, it remains to do a check of compatibility with the other constraints.

We now make a more detailed comparison between the results of this paper and three previous works, namely the algorithm of Gagnon, Hendren and Marceau, the framework of Knoblock and Rehof, and that of Agesen, Palsberg and Schwartzbach.

Gagnon, Hendren and Marceau The original motivation for this work was to try and improve on the performance of the algorithm proposed by Gagnon *et al.* [8]. Gagnon's work was a milestone in object-oriented type inference, providing the inference algorithm at the foundation of the widely used Soot framework. Our entire understanding of the problem was shaped by [8], and indeed we have adopted its framework of applying transformations to deal with bytecode that is verifiable yet cannot be typed statically.

The type inference phase of [8] works by constructing a graph of type constraints. This graph has two kinds of node, namely *hard* ones (which represent explicit types) and *soft* ones (representing type variables). An edge $a \leftarrow b$ means $a \leq b$ in our terms.

The construction of this graph is, in itself, quite expensive. It contains *all* type constraints induced by Jimple instructions. An important difference with our algorithm is that in first instance we only consider constraints that derive from assignments; and even those are not explicitly represented by a datastructure.

Solving the constraints means transforming the graph by merging soft nodes with hard nodes. Such a merging step is equivalent to inferring a type for a local variable. Merging can be done in the following three ways

- merge all elements of a connected component in the graph:

  $$a \leq b \wedge b \leq a \quad \Rightarrow \quad a \equiv b$$

- merge primitive types; if t is a primitive type then

  $$a \leq t \quad \Rightarrow \quad a \equiv t$$

- merge soft nodes that have only a single incoming edge (say from p) with p.

In addition to these merging steps, during the solution process one may remove transitive edges that are implied by others: if we have edges $a \leftarrow b \leftarrow c$, an edge $a \leftarrow c$ is redundant.

The above solution process is *sound*: when it succeeds, the result is a valid typing. It is however not *optimal* in the sense that there may exist a strictly smaller typing that is also valid. Some heuristics are applied, in particular in the choice of single-parent constraints to merge, to improve precision. These heuristics contrast sharply with the algorithm presented here, where there is a guarantee of optimality. In Section 6, we demonstrated that while it is rare for the algorithm of [8] to return suboptimal results, it does happen in practice. It is sometimes suboptimal for reference types, but more often for primitive types.

There is a price to pay for that optimality guarantee in our algorithm, however, and that is in the worst-case complexity. The algorithm of Gagnon *et al.* is clearly polynomial: the number of constraints is polynomial, and each step of the solution process is polynomial. By contrast, as we have indicated in Section 2, our algorithm can take exponential time. Indeed, in [8], it is argued that the problem of finding an optimal typing is NP-hard. However, as is argued there, for the type hierarchies found in practice, the exponential behaviour does not occur. The experiments in Section 6 confirm that observation.

Overall, our experiments in Section 6 also showed that the new algorithm presented here outperforms that of [8]. Furthermore, it is apparent from Figure 7 that the running time of the algorithm of Gagnon *et al.* is in practice cubic in the length of the method, whereas ours is linear. From the above discussion, the reasons for that performance difference are clear: our algorithm ellides the construction of a constraint graph. While the solution process of [8] is always quite costly, requiring the identification of strongly connected components, in our algorithm the most common case is two iterations of the fixpoint computation in the first stage.

Knoblock and Rehof A very thorough study of the problem of reconstructing types for local variables in Java bytecode was conducted by Knoblock and Rehof [15]. Like ourselves, they start with the observation that the problem is easily solvable if types form a lattice. They then go on to observe that there exists a smallest lattice in which the original type order is embedded, namely the *Dedekind-MacNeille completion*. Where the type inference algorithm finds a type that is not represented in the original program, a new type definition is generated.

This is quite different from the framework of Gagnon [8] where the code is sometimes transformed to make it typable, in the last resort by introducing casts. In [8] and in our setting, the type hierarchy itself is never modified. We believe that introducing new types is too drastic a structural change to the program to be allowed by an analysis and transformation framework, and that is definitely the case when used in decompilation.

The type elaboration algorithm has some similarities with that of [8] as well as ours, and it consists of the following steps:

1. First, all type constraints are collected. As emphasised earlier, we avoid the explicit representation of constraints, instead choosing to generate them on-the-fly from instructions.
2. Next, the set of type constraints is *closed* to take account of array types, again similar to [8]. In our setting, array types are treated in virtually the same way as other reference types (we have commented on our array-specific considerations in Section 3.)
3. The strongly connected components in the constraint set are collapsed, as in [8].
4. The new elements of the type hierarchy are introduced. This has no direct equivalent in our algorithm or that of [8], although at the level of primitive types, we do introduce a few fictitious types in the augmented value-set hierarchy of Figure 9.
5. The lattice algorithm is run to find a minimal typing. This is similar to Algorithm 1, the simple algorithm we started out with.
6. The solution is 'applied', possibly introducing unsafe narrowing conversions between primitive types. Obviously such unsafe narrowing conversions are undesirable, and should be avoided. While this step is dismissed as an afterthought in [15], it is a non-trivial contribution of the present paper to solve it carefully, as detailed in Section 5.

A small unpleasant issue is that sometimes introducing new types is not permissible according to the Java type system, because multiple inheritance between classes is not allowed. In the rare cases where the algorithm encounters such problems, it resorts to inferring the type *Object* and inserting casts.

Knoblock and Rehof report good experimental results for their algorithm, showing linear time growth of execution time against sizes of method bodies; in view of Figure 7 that compares very well against the algorithm of Gagnon *et al.*. The experiments in [15] are however rather less comprehensive than those reported here (and those in [8]), as they tested only 22,300 methods, against the 295,598 that we have experimented with. They do not report on tests that process bytecode that was not generated from Java source, which in our experience (and that of [8]) is crucial to test correctness, especially for the handling of primitive types. Furthermore there is no comparison in [15] of the quality of the typings against another algorithm, as we have done with [8] — there are many subtle issues that an implementation must handle, and it is very hard to get all the details right without at least one other algorithm to compare against. Unfortunately there is no publicly available implementation of [15], to compare its performance against the algorithm of [8], and the one presented here. In view of the complexity of the datastructures involved, it seems very unlikely, however, that it would outperform the very simple methods considered here. Furthermore, in [15], runtimes of up to 18s per method are reported for a method of 200 KB that is the result of a parser generator. Our algorithm processes methods of similar size and origin in 0.16s, which is much faster even allowing for a 10-fold speed increase in processors.

We remarked earlier in Section 2.2 that the use of sets of types, while predominant in the literature on object-oriented type inference, are inherently imprecise for certain examples where two types have multiple minimal common ancestors. In the worst case, this worsens the time complexity of our algorithm, but as we have shown, in practice the price for precision is not prohibitive.

Palsberg and Schwartzbach In a seminal paper [20], Palsberg and Schwartzbach laid the foundation for most subsequent work on type inference for object-oriented programs. A year later, they followed it up with various improvements (in particular to deal with collection types), and an efficient implementation [18].

The problem considered by Palsberg and Schwartzbach is more general and harder than the one considered here: given a program with no type annotations, infer the types of local variables and method signatures. Nevertheless, it is possible to make some observations about the connections between their work and the present paper.

The notion of types proposed in [20] is just a set of classes. However, prior to the type inference process, the inheritance hierarchy is expanded by augmenting each class with all the members it inherits from its supertypes, and making corresponding changes to statements in the code. As noted in [20], this flattening can result in a quadratic increase in the program size.

When presenting our algorithm, we already observed that using sets of types is not adequate because our aim is to obtain typings that are valid according to the Java type rules. Recall, however, that we did use upward-closed sets of typings. In fact, flattening the class hierarchy as Palsberg and Schwartzbach do corresponds to using upwards-closed sets of types:

$$S \leq T \quad \equiv \quad up(S) \supseteq up(T)$$

where $up(X) = \{ t \mid \exists s \in X : s \leq t \}$. In our implementation, we used small sets of representative elements to compute with upwards-closed sets. As a consequence we do not pay the cost associated with the expansion of the hierarchy in [20], which was again employed in the implementation paper [18].

Palsberg and Schwartzbach construct a graph representation of type constraints, named the *trace graph*. The nodes of the trace graph are methods, and the edges represent potential method invocations. Each node is decorated with a set of *local constraints*, which are precisely the constraints considered in the present paper. Additional constraints are attached on edges: these are different in nature from the simple type inequalities found as local constraints, instead being Horn clauses, relating assumptions about method arguments to method results.

Viewed in this light, it becomes clear that the algorithm we have presented here could be employed as a subroutine in a more general type constraint solver, doing the intraprocedural analysis required to solve local constraints. Indeed, as described in [18], one could construct the trace graph on demand, and whenever a new node is visited, we simply solve its local constraints with the new algorithm presented here. In [4], Agesen, Palsberg and Schwartzbach extend their approach to deal with dynamic and multiple inheritance. While the present paper has addressed multiple inheritance, we have not considered dynamic inheritance.

The algorithm of Gagnon *et al.* [8] is in fact more similar to that of Agesen, Palsberg and Schwartzbach than to our own, as it also operates on an explicitly constructed graph of constraints. The worst-case time complexity of our algorithm is worse than that of the constraints-based type inference algorithms. The complexity blow-up occurs because we maintain sets of typings, where a typing maps each variable to one type. Consequently, when (due to multiple inheritance) a variable x is given m types, and y is given n types, there are $m \times n$ typings maintained in our algorithm. This is not a problem in practice because while multiple inheritance is allowed in Java, its use is relatively rare. It is fair, therefore, to classify our approach to type inference as *optimising for the common case rather than the worst case.*

It is natural to wonder whether the principal idea underlying the present paper (process lowerbounds first to obtain minimal solution, then check upperbounds) can be applied to other constraint-based program analyses. While this seems likely, we have not yet investigated that question in any depth.

8. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm for local type inference, and measured its performance in inferring types for local variables in Java bytecode. Our algorithm outperforms the previously best available solution for that problem (due to Gagnon *et al.* [8]), especially on methods that contain many statements. Not only does it exhibit better runtime efficiency, its results are also guaranteed to be optimal, in the sense that a tightest possible typing is returned. The key design principle we have used is to optimise for the common case, rathan for the worst case. In particular, while our algorithm handles multiple inheritance, it exploits the observation that in practice, multiple inheritance is used relatively infrequently.

At a more technical level, one key idea is to first process type constraints that impose lowerbounds (*i.e.* constraints from assignments), and find minimal solutions to those. Next, in a second phase, we check use constraints (which all impose upperbounds) and prune out minimal solutions that do not satisfy those additional constraints. This turns out to be very effective if the goal is to infer types according to the type rules of Java bytecode. For use in a typed intermediate language with the aim of optimisation, this is the type inference algorithm to chose.

However, as pointed out by a number of previous works [8, 15, 17, 22], when the purpose is decompilation, the requirements on type inference are somewhat different. Here we must consider the type hierarchy not as it is dictated by bytecode, but as it is given by the Java source language. The discrepancy lies in the way primitive types are treated: for example *boolean*, *byte* and *char* are incomparable in source, but all represented by integers at bytecode level. We have shown how to handle that problem by augmenting the checking phase of our algorithm to make appropriate adjustments. The cost of having to do this is significant, however, sometimes taking up to 47% longer. It is therefore recommended that when the purpose is optimisation and not decompilation, the bytecode type hierarchy is used instead.

The main item of future work is to examine the impact of this local type inference algorithm in the context of interprocedural type inference, in particular for the efficient construction of call graphs. Another is to examine other applications of constraint-based program analysis, and whether the method given here can be generalised to such other analysis problems.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[2] Ole Agesen. The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism. In Walter G. Olthoff, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995.

[3] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1996. Sun Microsystems, Technical report TR-96-52.

[4] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Software Practice and Experience*, 25(9):975–995, 1995.

[5] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Implementation of our local type inference algorithm (including experiments). `http://musketeer.comlab.ox.ac.uk/typeinference/`, 2008.

[6] Eric Bodden. A denial-of-service attack on the java bytecode verifier. `http://www.bodden.de/research/javados/`, 2008.

[7] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *Object-Oriented Programming, Systems and Languages*, pages 15–34, 2004.

[8] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 199–219, 2000.

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[10] Justin Owen Graver. *Type-Checking and Type Inference for Object-Oriented Programming Languages*. PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[11] Justin Owen Graver and Ralph E. Johnson. A type system for Smalltalk. In *Symposium on Principles of Programming Languages (POPL)*, pages 136–150. ACM Press, 1990.

[12] Andreas V. Hense. *Polymorphic Type Inference for Object-Oriented Programming Languages*. PhD thesis, Universität des Saarlandes, 1994.

[13] Eric J. Holstege. *Type Inference in a Declarationless, Object-Oriented Language*. PhD thesis, California Institute of Technology, 1982. Technical report 5035.

[14] Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, 6:133–134, 1928.

[15] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for java bytecode. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):243–272, 2001.

[16] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[17] J. Miecnikowski and L. J. Hendren. Decompiling java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag, 2002.

[18] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349, 1992.

[19] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995.

[20] Jens Palsberg and Michael I. Schwartzbach. Object-oriented

type inference. In Andreas Paepcke, editor, *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 146–161. ACM Press, 1991.

[21] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 324–340. ACM Press, 1994.

[22] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *Conference on Object-Oriented Technologies (COOTS)*, pages 185–198, 1997.

[23] Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16:23–96, 1978.

[24] Steven Alexander Spoon. *Demand-driven type inference with subgoal pruning*. PhD thesis, Georgia Institute of Technology, 2005.

[25] Norihisa Suzuki. Inferring types in smalltalk. In *Symposium on Principles of Programming Languages*, pages 187–199. ACM Press, 1981.

[26] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.