Efficient Inference of Static Types for Java Bytecode*

Etienne M. Gagnon, Laurie J. Hendren and Guillaume Marceau

Sable Research Group, School of Computer Science
McGill University, Montreal, Canada
[gagnon,hendren,gmarceau]@sable.mcgill.ca

Abstract. In this paper, we present an efficient and practical algorithm for inferring static types for local variables in a 3-address, stackless, representation of Java bytecode.
By decoupling the type inference problem from the low level bytecode representation, and abstracting it into a *constraint system*, we show that there exists verifiable bytecode that cannot be statically typed. Further, we show that, without transforming the program, the static typing problem is NP-hard. In order to get a practical approach we have developed an algorithm that works efficiently for the usual cases and then applies efficient program transformations to simplify the hard cases.
We have implemented this algorithm in the Soot framework. Our experimental results show that all of the 17,000 methods used in our tests were successfully typed, 99.8% of those required only the first stage, 0.2% required the second stage, and no methods required the third stage.

1 Introduction

Java bytecode is rapidly becoming an important intermediate representation. This is predominately because Java bytecode interpreters and JIT-compilers are becoming more common, and such interpreters / compilers are now a standard part of popular web browsers. Thus, Java bytecode (henceforth referred to as simply bytecode) has become a target representation for a wide variety of compilers, including compilers for Ada [23], ML [14], Scheme [5], and Eiffel [20].

Bytecode has many interesting properties, including some guarantees about verifiable bytecode that ensure that verified bytecode programs are *well-behaved.* For example, verifiable bytecode guarantees that each method invocation has the correct number and type of arguments on the Java expression stack. Verification is done partly statically via a flow analysis of the bytecode, and partly via checks that are executed at runtime. As part of the static verification, a flow analysis is used to estimate the type of each local variable and each location on the expression stack, for each program point. However, as we will show in section 3 this is **not** the same typing problem as the one addressed in this paper.

Although bytecode has many good features, it is not an ideal representation for program analysis / optimization or program understanding. For analysis /

* This work has been supported in part by FCAR and NSERC.

optimization, the expression stack complicates both the analyses and subsequent transformations. In addition, the stack-based representation does not map nicely to real register-based architectures. For these sorts of optimizing compiler applications a more traditional three-address code is preferable, and is used in many optimizing Java compilers. For program understanding, the bytecode is too low-level, and one would like to present a higher-level view to the programmer. One example of a high-level representation is decompiling bytecode back to Java. Note that to be generally useful such decompilers should work for any verifiable bytecode, not just bytecode produced by Java compilers.[1]

When bytecode is translated to a three-address representation or high-level representation it is important that all variables should be given a static type that is correct for all uses of that variable. For a decompiler, each variable needs to have a declared type that is type correct for all uses of that variable. For three-address representations, the type of a variable can be used to improve analysis and optimization. We have found having types for local variables to be indispensable in our compiler, and one example use, improving the quality of the call graph, is presented in section 7.2.

In this paper, we address the problem of inferring a *static type* for each variable in a three-address representation of bytecode called Jimple[26, 25]. Jimple is part of the Soot compiler framework that is used for both compiler optimizations and decompilation. It is a fairly standard representation, so our results should apply to other similar representations.

In order to give a feel for the problem, consider the simple example in Figure 1. Figure 1(a) gives an untyped method in a Jimple-like intermediate representation. Note that there is some type information which comes directly from the bytecode. For example, the signature of method `f` is specified in the bytecode, so we know a fixed type for the return value, and we know some type information from `new` instructions. However, local variables, such as `a`, `b`, `c` and `s` do not have an explicit type in the bytecode. We can determine correct types for these variables by collecting type constraints. Figure 1(b) shows the class hierarchy, and figure 1(c) shows the constraints imposed by each statement. We formulate the typing problem as a graph problem. Figure 1(d) shows a graph that represents both the class hierarchy and the type constraints on the variables. Types in the hierarchy are shown as double circles which we call *hard nodes*, while type variables are shown as single circles which we call *soft nodes*. A solution to the typing problem is found by coalescing nodes together. If nodes can be coalesced so that each coalesced node contains exactly one hard node, then we have found a solution to the typing problem. Figure 1(e) shows one possible coalescing of the graph, and this corresponds to the typed method in Figure 1(e). Note that there may be more than one correct solution. For this example another correct solution would be to assign `a`, `b` and `c` the type `Object`. In general, we prefer

[1] Also note that by combining a compiler that translates from a high-level language `X` to bytecode with a decompiler from bytecode to Java, one has a tool for translating from `X` to Java.

a typing that gives more specific types since this will help more in subsequent analyses.

```
public java.lang.String f()
{   <unknown> a;
    <unknown> b;
    <unknown> c;
    <unknown> s;

s1: c = new C();
s2: b = new B();
    if ( ... )
s3:   a = c;
    else
s4:   a = b;
s5: s = a.toString();
s6: return(s);
}
```

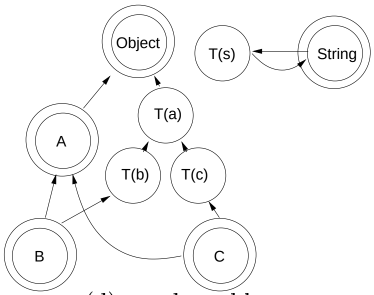
(a) untyped method

```
class A extends Object
{ ... }

class B extends A
{ public String toString() ...;
  ...
}

class C extends A
{ public String toString() ...;
  ...
}
```

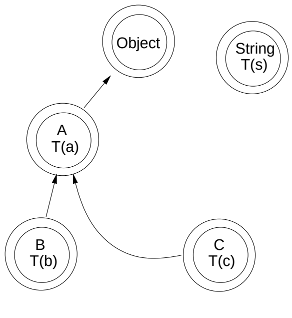
(b) class hierarchy

s1: $T(c) \leftarrow C$
s2: $T(b) \leftarrow B$
s3: $T(a) \leftarrow T(c)$
s4: $T(a) \leftarrow T(b)$
s5: $Object \leftarrow T(a)$
$T(s) \leftarrow String$
s6: $String \leftarrow T(s)$

(c) constraints



(d) graph problem



(e) solution

```
public java.lang.String f()
{   A a;
    B b;
    C c;
    java.lang.String s;

s1: c = new C();
s2: b = new B();
    if ( ... )
s3:   a = c;
    else
s4:   a = b;
s5: s = a.toString();
s6: return(s);
}
```

(f) typed method

Fig. 1. Simple example of static typing

The type inference problem would seem easy at first glance, and for our simple example it would be quite easy to deduce types during the bytecode to Jimple translation. However, there are three points that make the general typing problem difficult: (1) the program point specific nature of the bytecode verification, (2) multiple inheritance due to interfaces, and (3) the correct typing of arrays. In fact, we will show that the type inference problem is NP-hard. However, we propose an efficient, polynomial time, multi-stage algorithm that bypasses this complexity by performing program transformations that simplify the type inference problem, without affecting program semantics, when a difficult case is encountered. Our algorithm performs two kinds of transformations: (1) variable splitting at object creation sites, and (2) insertion of type casts that are guaranteed to succeed at run-time. Our experimental results show that all of the 16,492 methods extracted from 2,787 JDK 1.1 and SPEC jvm98 classes were typed by our algorithm, without inserting any type casts. Variable splitting was only applied in 29 methods.

It is important to contrast this work, where we find a *static* type consistent with *all* uses of a variable, with other type inference analyses where the main focus is to statically infer the set of dynamic (or concrete) types that a variable could hold, at a particular program point at run-time [17, 18, 1, 2]. We will call such algorithms *run-time type analysis* to distinguish them from our *static-type analysis*. For our example program in Figure 1(a), run-time type analysis would infer that the variable `a` at program point `s1` could have type `B`, whereas at program point `s5` `a` could have types {`B`, `C`}. In our typing problem we need to find **one** static type that is consistent with **all** uses of `a`. As we show in section 7.2, our static type is actually a reasonably good starting point for other analyses, including a run-time type analysis we have built on top of typed Jimple.

Our paper is structured as follows. In section 2 we present our three-address representation. In section 3 we show some examples to demonstrate why this typing problem is difficult. In section 4, we define the general static type inference problem, and give the main algorithm for programs without arrays. In section 5 we present extensions to our algorithm to handle arrays. In section 6 we show how to infer integer types. Section 7 contains our experimental results. Finally, we review related work in section 8 and present our conclusions in section 9.

2 A 3-Address Representation: Jimple

We assume that the reader is already familiar with Java bytecode. A complete description of the *class file format* can be found in [13]. Furthermore, we assume that all analyzed bytecode would be successfully verified by the Java bytecode verifier[13]. It is important to remember that the verifiability of the code implies that it is *well behaved*, but it does not imply that it is *well typed*.

While the bytecode format seems of great interest for implementing an interpreter, it is not well suited for reasoning about bytecode, since many operands are on the stack and thus do not have explicit names. In order to alleviate this difficulty, many Java optimizing compilers convert bytecode to a more traditional 3-address-code, where all stack-based operations are transformed into local variable based operations. This is made possible by the conditions met by verified bytecode, most notably: the constant stack depth at each program point, and the explicit maximum depth of stack and number of local variables used in the body of a method.

The bytecode to 3-address-code transformation is done by computing the stack depth at each program point, introducing a new local variable for each stack depth, and then rewriting the instruction using the new local variables[2].

For example:

```
iload_1    (stack depth before 0 after 1)
iload_2    (stack depth before 1 after 2)
iadd       (stack depth before 2 after 1)
istore_1   (stack depth before 1 after 0)
```

is transformed into:

```
stack_1 = local_1
stack_2 = local_2
stack_1 = stack_1 iadd stack_2
local_1 = stack_1
```

In producing the 3-address-code it is simple to retain all type information contained in bytecode instructions. So, for instance, every virtual method contains the complete signature of the called method, as well as the name of the class declaring the method. However, as there are no explicit types for locals or stack locations, it is more difficult to find types for these variables. In our compiler we produce a 3-address representation called Jimple, that is first created in an untyped version, where the types of local variables are unknown. Every verifiable bytecode program has an equivalent *untyped* Jimple representation.

In final preparation, prior to applying the typing algorithms outlined in this paper, a data flow analysis is applied on the Jimple representation, computing definition-use and use-definition (*du-ud*) chains. Then, all local variables are split into multiple variables, one for each web of du-ud chains. Our example would be transformed to:

```
stack_1_0 = local_1_0
stack_2_0 = local_2_0
stack_1_1 = stack_1_0 iadd stack_2_0
local_1_1 = stack_1_1
```

Note that `stack_1` has been split into `stack_1_0` and `stack_1_1`, and similarly `local_1` has been split into `local_1_0` and `local_1_1`. This splitting is quite important, because a single local or stack location in the bytecode can refer to different types at different program points. This form of Jimple looks overly long, with many spurious copy statements. In our framework the code is cleaned up using standard techniques for copy propagation and elimination.

3 Challenges of Typing

The static typing problem looks quite simple at first, but there are subtle points that make the problem difficult. In this section we illustrate some difficulties by

[2] In reality, the stack analysis, the introduction of new local variables, and the transformation are not as straight-forward as it looks here. This is due to the presence of subroutines (the *jsr* bytecode instruction) and double-word values (*long*, *double*). A complete description of the bytecode to Jimple transformation can be found in [26, 25].

showing differences between the typing problem for a 3-address representation with local variables, and the typing approximation done by the Java verifier. Another subtle point is how to deal with arrays, and this is dealt with in Section 5.

3.1 Declared variable types versus types at program points

Part of the Java verifier is a flow analysis that estimates, at each program point, the type of values stored in each local variable and each stack location. This type information is used to ensure that each bytecode instruction is operating on data of the correct type. In our typing problem we wish to give a type to each variable that is correct for **all** uses and definitions of that variable (i.e. the same type must be correct at multiple program points).

Consider Figure 2 where two methods `hard` and `harder` illustrate the point. In method `hard`, the Java verifier would infer that `x` has type `CA` at program point `s1` and type `CB` at program point `s2`. For program point `s3` the verifier merges the types from each branch by taking their closest common superclass, which is `Object`. Thus, for three different program points, the verifier has three different types. However, for our problem, we want to assign one type to local variable `x`. In this case, it is possible to satisfy all constraints and assign type `Object` to variable `x`. However, to find consistent types the whole method must be analyzed, the types cannot be computed "on-the-fly" as is done in the verifier.

Now consider method `harder` in Figure 2. This is similar to the previous case, but now it is not possible to give a single static type to variable `y`. At program point `s1` `y` **must** have type `CA` and at program point `s2` `y` **must** have type `CB`. In order to statically type this program, it must be transformed to include extra copy statements (as one would get by translating from an SSA form) or by introducing type casts. Note that one would not see the `harder` case in bytecode produced from an ordinary Java compiler, however we have seen cases like this in bytecode produced by compilers for other languages.

```
class CA extends Object { f(){...} ... }
class CB extends Object { g(){...} ... }
class MultiDef extends Object
{ void hard()
  { <untyped> x;
    if( ... )
s1:   x =  new CA();
    else
s2:   x =  new CB();
s3: x.toString();
  }
```

```
  void harder()
  { <untyped> y;
    if( ... )
s1:   { y = new CA(); y.f(); }
    else
s2:   { y = new CB(); y.g(); }
s3: y.toString();
  }
}
```

Fig. 2. Multiple definition and use points

3.2 Problems due to interfaces

Interfaces in Java give a restricted form of multiple inheritance, and this leads to problems in finding a static typing in some cases. Consider the example in Figure

3(a), where the class hierarchy is defined as in Figure 3(b). At program point `s1` `aa` has interface type `IC`, and at program point `s2` `aa` has interface type `ID`. The difficulty comes at the merge point because there is no single superinterface for `IC` and `ID`, rather there are two unrelated choices, `IA` and `IB`. The Java verifier will choose the type `Object`, and then check the invokeinterface calls at **runtime**. These runtime checks will pass, and so from the verification point of view, this program is well-behaved.

```
class InterfaceDemo
{ IC getC() { return new CC(); }
  ID getD() { return new CD(); }

  void hardest()
  { <untyped> aa;

    if( ... )
s1:   aa = getC();
    else
s2:   aa = getD();

s3: aa.f(); // invokeinterface IA.f
s4: aa.g(); // invokeinterface IB.g
  }
}
```

(a) untyped program

```
class CC implements IC
{ void f() {}
  void g() {}
}

class CD implements ID
{ void f() {}
  void g() {}
}

Interface IA { void f(); }
Interface IB { void g(); }
Interface IC extends IA, IB {}
Interface ID extends IA, IB {}
```

(b) hierarchy

Fig. 3. Typing interfaces

Now consider our problem of finding one static type for `aa`. In this case there is no solution, even though the bytecode is verifiable. If we chose type `IA`, then the type at statement `s4` is wrong, if we chose type `IB`, the type at statement `s3` is wrong, if we chose type `IC`, the type at statement `s2` is wrong, and if we chose type `ID`, the type at statement `s1` is wrong. In fact, one can not write a Java program like this Jimple program and give a correct static type to `aa`. However, remember that our Jimple code comes from bytecode produced from any compiler or bytecode optimizer, and so this situation may occur in verifiable bytecode.

One might be tempted to think that adding extra copies of the variable, like in SSA form would solve this problem as well. However, if we rewrite 3(a) in SSA form, we get:

```
    if( ... )
s1:   aa1 = getC();
    else
s2:   aa2 = getD();

s3a: aa3 = phi(aa1, aa2);
s3: aa3.f(); // invokeinterface IA.f
s4: aa3.g(); // invokeinterface IB.g
```

Clearly this does not solve the problem, there is still no type solution for `aa3`.

4 Three-Stage Algorithm

4.1 Algorithm overview

The goal of the typing algorithm is to find a static type assignment for all local variables such that all type restrictions imposed by Jimple instructions on their arguments are met. In order to solve this problem, we abstract it into a *constraint system*. For convenience of implementation (and description), we represent this constraint system as a directed-graph.

We initially restrict our type inference problem to programs that do not include arrays, nor array operations. This allows us to illustrate the constraint system.

Finding whether there exists or not a static-type assignment that solves this constraint system is similar to solving the UNIFORM-FLAT-SSI problem, which Tiuryn and Pratt have shown to be NP-Complete[24]. Thus, the overall typing problem is NP-Hard.

Given this complexity result, we have chosen to design an efficient algorithm that may perform program transformations to make the typing problem simpler. We first give an overview of our algorithm, and then describe each stage in more detail.

An efficient 3-stage algorithm The algorithm consists of three stages. The first stage constructs a directed-graph of program constraints. Then, it merges the connected components of the graph, and removes transitive constraints. Finally, it merges single constraints. At this point, it succeeds if all variables have valid types, or it fails if a variable has no type, or if a type error was detected in the process.

If the first stage fails to deliver a solution, the second stage applies a variable splitting transformation, and then reruns stage 1 on the transformed program. We have only found one situation where variable splitting is required, and that is for variables which are assigned new objects (i.e. for statements of the form `x = new A()`).

If stage 2 fails, then stage 3 proceeds as follows. A new constraints graph is built, where this graph only encodes *variable definition* constraints. In this graph, *variable use* constraints are not recorded, and interface inheritance is ignored. In other words, each interface has a single parent `java.lang.Object`. Then, the constraints system is solved using the *least common ancestor* LCA of classes and interfaces (which is now always unique). Once all variables are assigned a type, *use constraints* are checked on every original Jimple statement, and type casts are added as needed to satisfy the constraints. The verifiability of the original program guarantees that these inserted casts will always succeed at run-time.

Handling arrays This section describes the basic constraint system for programs without arrays. We extend the constraint system, with extra notation for array constraints, in Section 5. We then show how to transform an array problem into a *restricted problem* (with no array constraints), and how to propagate the solution of the restricted problem back to the original array problem.

Implementing the algorithm We have implemented the algorithm, but in this paper we do not discuss implementation details. It is quite straightforward to achieve a simple implementation using efficient algorithms for strongly-connected components and fast union on disjoint sets [6].

4.2 Stage 1

Constraint system In this section, we show how to transform the type inference problem into a constraint system represented as a directed graph. Intuitively, the graph represents the constraints imposed on local variables by Jimple instructions in the body of a method. In this initial version, we assume that the analyzed Jimple code contains no arrays and no array operations. Further, we infer primitive types as defined for Java bytecode [13]. In particular, *boolean*, *byte*, *short*, and *char* are all treated as *int*. Section 6 presents an algorithm that can be used to infer these different integer types.

The *constraint graph* is a directed graph containing the following components:

1. *hard node*: represents an explicit type;
2. *soft node*: represents a type variable; and
3. *directed edge*: represents a constraint between two nodes.

A directed edge from node b to node a, represented in the text as $a \leftarrow b$, means that b should be *assignable* to a, using the standard *assignment compatibility* rules of Java [13, 10]. Simply stated, b should be of the same type as a, or a should be a *superclass* (or *superinterface*) of b.

The graph is constructed via a single pass over the Jimple code, adding nodes and edges to the graph, as implied by each Jimple instruction. The collection of constraints is best explained by looking at a few representative Jimple statements. We will look at the simple assignment statement, the assignment of a *binary* expression to a local variable, and a virtual method invocation. All other constructions are similar.

A *simple assignment* is an assignment between two local variables $[a = b]$. If variable b is assigned to variable a, the constraints of *assignment compatibility* imply that $T(a) \leftarrow T(b)$, where $T(a)$ and $T(b)$ represent the yet unknown respective types of a and b. So, in this case, we need to add an edge from $T(b)$ to $T(a)$ (if not already present). This is shown in figure 4.

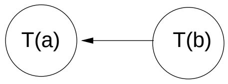


Fig. 4. b assigned to a

An assignment with a more complex right-hand-side results in more constraints. For example, the statement $[a = b + 3]$, generates the following constraints: $T(a) \leftarrow T(b)$, $T(a) \leftarrow int$, and $int \leftarrow T(b)$.

Our last and most complicated case is a method invocation, where constraints are generated for the receiver, the actuals, and the variable on the left-hand-side. For example, consider $[a = b.equals(c)]$, or with the full type signature: $a =$ *virtualinvoke b.[boolean java.lang.Object.equals(java.lang.Object)] (c)*. We get the following constraints, each involving a hard node: (1) $java.lang.Object \leftarrow T(b)$, from the declaring class of *equals*; and (2) $java.lang.Object \leftarrow T(c)$, from the argument type in the method signature; and (3) $T(a) \leftarrow int$, because the return type of *equals* is *boolean* (we have a single integer type).

As shown in figure 1, our type inference problem now consists of merging soft nodes with hard nodes, such that all *assignment compatibility* constraints, represented by edges, are satisfied. Merging a soft node with a hard node is equivalent to inferring a type for a local variable. If no such solution exists (or it is too costly to find), or if a node needs more than one associated type (e.g. a soft node is merged with two or more hard nodes), then the first stage of the inference algorithm fails.

Connected components Our first transformation on the constraint graph consists of finding its connected components (or cycles). Every time a connected component is found, we merge together all nodes of connected component, as illustrated in figure 5.

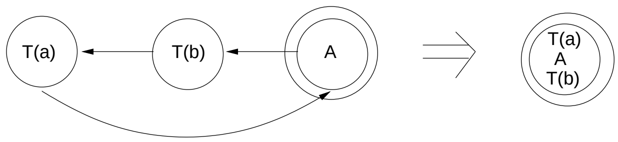


Fig. 5. Merging connected components

This is justified because every node in a connected component is indirectly *assignable* to and from any other node in the same connected component. It follows that all these nodes must represent the same type, in any solution to the type inference problem.

We can divide connected components into three kinds. First, there are connected components without hard nodes. In this case, nodes are simply merged, and all constraints of all nodes are propagated to the representative node[3]. Second, some connected components have a single hard node. In this case, all soft nodes are merged with the hard node, then all constraints are verified. If any constraint can't be satisfied, the first stage of the algorithm fails. Third, it may be that a connected component has two or more hard nodes. When this occurs, the first stage fails.

In this step, we also take advantage of the verifier restrictions on primitive types to merge respectively all values in a *transitive relation* with any of the primitive types: *int, long, float*, and *double*. Figure 6 shows an example of primitive type merge. It is enough that a node be indirectly assignable to *or* from a primitive type hard node to be merged with it. This is because there is no automatic conversion between primitive types.

Transitive constraints Once the connected components are removed from the constraint graph, we are left with a directed-acyclic-graph (DAG). Our

[3] Constraints from the representative node to itself are eliminated.

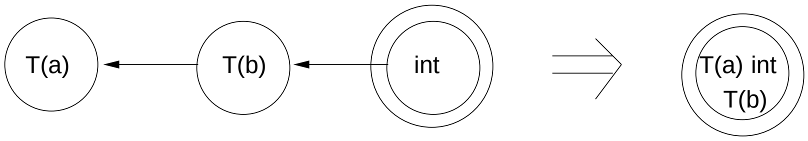


Fig. 6. Merging primitive types

next transformation consists of removing redundant constraints (edges) from this DAG by eliminating any transitive constraints in the graph. A transitive constraint from a node y to a node x, is a constraint $x \leftarrow y$ such that there exists another constraint $p \leftarrow y$ where p is not x and there is a path from p to x in the directed graph.

Transitive constraints are removed regardless of the kind of nodes involved (soft, hard), with the exception of hard-node to hard-node constraints[4]. This is shown in figure 7.

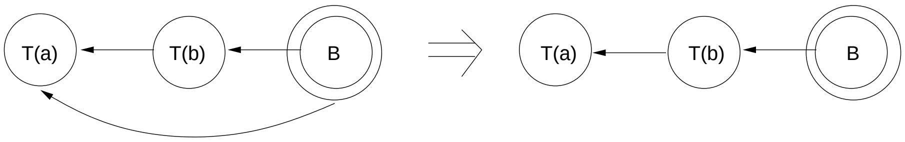


Fig. 7. Removing transitive constraints

Single Constraints Nodes that have only one parent or one child constraint can be simplified. A node x is said to have a *single parent constraint* to a node y, if $y \leftarrow x$ and for any $p \neq y$ there is no constraint $p \leftarrow x$. A node x is said to have a *single child constraint* to a node y, if $x \leftarrow y$ and for any $p \neq y$ there is no constraint $x \leftarrow p$.

Our next transformation consists of merging soft nodes that have single constraints to other nodes. To improve the accuracy of our results, we do this using the following priority scheme:

1. Merge single child constraints: Merge x with y when x is a soft node with a single child constraint to any other node y. (Merging with children results in lower (thus more precise) types in the type hierarchy).
2. Merge with least common ancestor: This is a special case. When x is a soft node that only has child constraints to hard nodes representing *class types*, we can safely replace these constraints by a single child constraint to the hard node representing the least common ancestor of the class types involved. Then we can merge the resulting single child constraint.
3. Merge single soft parent constraints: Merge x with y when x is a soft node with a single parent constraint to another *soft node* y.
4. Merge remaining single parent constraints: Merge x with y when x is a soft node with a single parent constraint to another node y.

Examples of this are shown in Figures 1 and 8.

[4] Hard-node to hard-node constraints represent the type hierarchy.

When a soft node has no explicit parent, we can safely assume that it has the hard node representing *java.lang.Object* as parent. We also introduce (as does the verifier) a *null* type, which is a descendant of all reference types. When a soft node has no child, which means that it was never *defined*, we assume that it has *null* as a child.

Stage 1 succeeds if all soft nodes are merged with hard nodes at the end of this step. It fails when merging a soft node with a hard node exposes an invalid constraint, or when there remains a soft node at the end of the step.

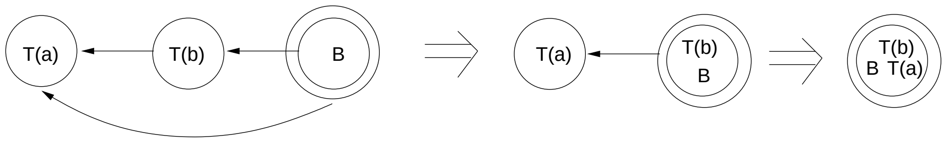


Fig. 8. Merging single constraints

4.3 Stage 2

In some cases, stage 1 fails to deliver a solution. In our experiments, this only happened in cases similar to the problem exposed in method `harder` of Figure 2. More precisely, the source of the problem is that Java and other languages use a simple `new` expression to both create and initialize a new object, whereas in bytecode, the same operation is done in two separate steps: the object is created using the `new` bytecode, but it is then initialized by invoking the <`init`> method on the newly created object. This is shown in Figure 9, where the method called `java` shows the method as it would appear in Java, and the method called `three_address` shows the extra `<init>` instructions that are exposed at the bytecode level.

```
class CA
  extends Object
  { ... }
class CB
  extends Object
  { ... }
class MultiDef
  extends Object
{ void java()
  { Object y;
    if( ... )
      y = new CA();
    else
      y = new CB();
    y.toString();
  }
}
```

```
void three_address()
{ <untyped> y;
  if( ... )
    { y =  new CA();
      y.[CA.<init>()]();
    }
  else
    { y =  new CB();
      y.[CB.<init>()]();
    }
  y.toString();
}
```

```
void fixed_three_address()
{ <untyped> y, y1, y2;
  if( ... )
    { y1 =  new CA();
      y = y1;
      y1.[CA.<init>]();
    }
  else
    { y2 =  new CB();
      y = y2;
      y2.[CB.<init>]();
    }
  y.toString();
}
```

Fig. 9. Object creation in Java versus 3-address code

In stage 2, we solve this problem by introducing copy statements at every object creation site. This is shown in Figure 9 in the method `fixed_three_address`. After inserting the extra copy statements, we simply reapply stage 1.

Experimental results show us that this very simple transformation is very effective at solving all type inference difficulties found in code generated from normal compilers.

4.4 Stage 3

It is possible that the previous stages fail and this would happen with the method `hardest` in Figure 3. However, current compilers and programs seem not to expose such difficult cases. In the future, optimizing compilers could get more aggressive, and programmers might start designing more elaborate interface hierarchies. In order to provide a complete, yet efficient type inference algorithm, we designed this third stage.

First, we note that, due to the *good behavior* guarantees provided by the bytecode verifier, a very crude solution to all type inference problems would be to separate local variables into sets of: reference, double, float, long and int variables. Then, assign the type *java.lang.Object* to all reference variables. Finally, introduce type casts at every location where *use constraints* would be violated. All the introduced casts are guaranteed to succeed at runtime.

But this solution would be somewhat useless in the context of an three-address code optimizer, as type information would be too general, and it would not offer much information in decompiled Java programs.

Our solution also depends on the *good behavior* of verifiable bytecode, but offers a much improved type accuracy without sacrificing simplicity. We simply rebuild the constraint graph without *use constraints*[5], and we ignore the interface hierarchy by assuming that all interfaces have a single parent *java.lang.Object*. In this hierarchy, every two types have an LCA.

As in stage 1, we merge strongly connected components. But, in this *definition constraints* graph, all hard-node to soft-node constraints are parent constraints. So, no strongly connected component contains a hard node. Thus, this step will not detect any type error.

Then, as in stage 1, we eliminate transitive constraints and merge single constraints. When merging single constraints, we replace multiple child constraints from a soft node to hard nodes by a single child constraint from the soft node to the node representing the LCA type of classes and interfaces involved. Unlike stage 1, this is guaranteed to deliver a solution.

The type assignment of this solution may violate some use constraints. So, in a last step, we check every three-address statement for constraint violations, and introduce type casts as needed.

Figure 10 shows this solution applied to the examples in Figure 2 (`harder`), and Figure 3 (`hardest`).

[5] A *definition constraint* is a constraint imposed by a definition e.g. $x = new\ A$ is a definition of x, and so it introduces a constraint that would be included in this graph. A *use constraint* in imposed by a use of a variable e.g. *return(x)* uses x and so any constraints imposed by this use would not be included in the graph.

```
   void harder()
   { Object y;
     if( ... )
s1:    { y =  new CA();
         ((CA) y).f(); }
     else
s2:    { y =  new CB();
         ((CB) y).g(); }
s3: y.toString();
   }
```

(a) Figure 2 (`harder`) solution

```
   void hardest()
   { Object aa;

     if( ... )
s1:    aa = getC();
     else
s2:    aa = getD();

s3: ((IA) aa).f(); // invokeinterface IA.f
s4: ((IB) aa).g(); // invokeinterface IB.g
   }
```

(b) Figure 3 (`hardest`) solution

Fig. 10. Adding casts

5 Array Constraints

To infer types in programs using arrays, we introduce array constraints in the constraints graph. An *array constraint* represents the relation between the type of an array and the type of its elements. We write $A \mapsto B$, to indicate that B is the element type of array type A (or more simply, A is an array of B). In graphs, we represent these constraints using dashed directed edges from the array type to the element type.

This simple addition allows us to collect constraints for all three-address statements. For example, the program fragment `a[d] = b; c = a;` generates the following constraints: $a \mapsto b, d \leftarrow int$, and $c \leftarrow a$.

In Java bytecode, $(A[] \leftarrow B[])$ *iff* $(A \leftarrow B)$ and $(A \leftarrow B[])$ *iff* $(A \epsilon \{Object, Serializable, Cloneable\})$. We take advantage of this to build an equivalent constraints graph without any array constraints. Then we solve the new problem using the algorithm presented in Section 4. Finally, we use this solution to infer the type of arrays, reversing our first transformation. This transformation is applied in both stage 1 and stage 3, if needed.

We now give a more detailed description of this process. First, we compute the array depth of soft nodes in the constraints graph using a work list algorithm and the following rules:

- Every hard node has an array depth equal to the number of dimensions of the array type it represents, or 0 if it is not an array type.
- *null* has array depth ∞. (*null* is descendant of all types, including array types of all depth[6]).
- A soft node with one or more child constraint has an array depth equal to the smallest *array depth* of its children.
- A soft node with an array constraint has a depth equal to one + the depth of its element node.
- When we verify our solution, a soft node with one or more parent constraints must have an array depth greater or equal to the greatest *array depth* of its parents[7]. (This rule is not used in stage 3).

[6] We could also use 256, as Java arrays are limited to 255 dimensions.
[7] If this rule fails, stage 1 fails.

We merge with *null* all soft nodes with array depth equal to ∞. Then, we complete the constraints graph by adding all missing array constraints (and soft nodes) so that every node of array depth n greater than 0 (called *array node*) has an array constraint to a node of array depth $n-1$.

The final step in the transformation is to change all constraints between *array soft nodes* and other nodes into constraints between *non-array nodes* using the following rules:

- Change a constraint between two nodes of equal depth into a constraint between their respective element nodes.
- Change a constraint between two nodes of different depth into a constraint between the element type of lowest depth node and $java.lang.Cloneable$ and $java.io.Serializable$.

This is illustrated in figure 11.

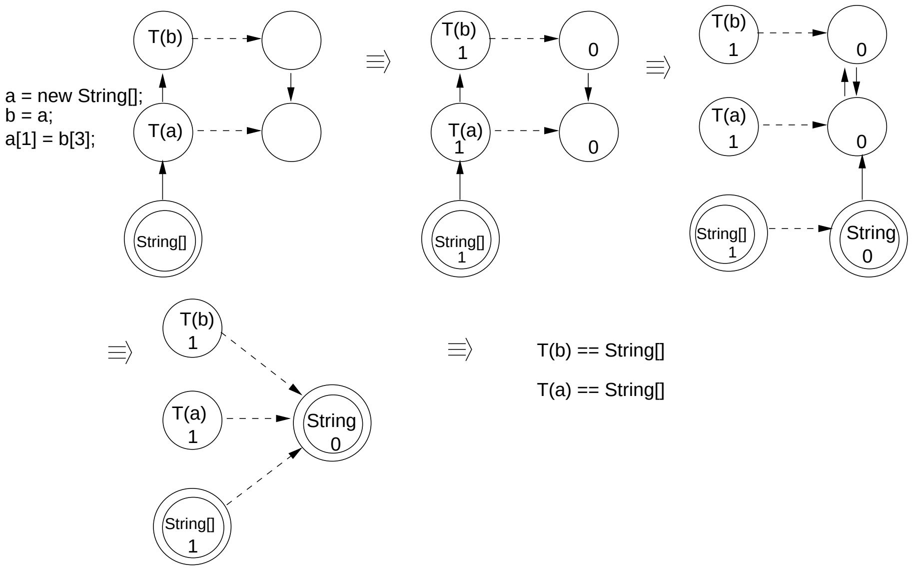


Fig. 11. Solving array constraints

Then we use the algorithm of section 4 to solve the typing problem on the graph of non-array nodes. Then we use this solution to infer the type of array nodes. For example, if $x \mapsto y$, and $y = A$, then $x = A[]$

In order to correctly handle the case of primitive array types (*boolean*[]...[], *short*[]...[], *char*[]...[], *byte*[]...[]), we merge these hard nodes with all their same-depth neighbors before constraints propagation[8].

6 Integer Types

While the algorithm presented in previous sections infers the necessary types for optimizing three-address code, these types are not sufficient for Java decompilers.

[8] This is necessary because the depth 0 for all these types is *int*.

All `boolean, byte, short, char` values are automatically operated upon as `int` values by the bytecode interpreter. Furthermore, the Java verifier does not check for consistent use of these types.

It is thus possible to construct bytecode programs with dubious semantics as:

```
boolean erroneous(int a)   // boolean return value
{ return a;  // valid bytecode!
}
void dubious()
{ <unknown> b = erroneous(5);
  System.out.[void println(int)](b);  // prints 1 or 5?
}
```

We developed an algorithm that infers the basic types `boolean, byte, short, char, int` for all variables that are assigned an `int` type by the initial 3-stage algorithm.

This algorithm operates in two stages. The first stage uses the type hierarchy in Figure 12(a), and consists of:

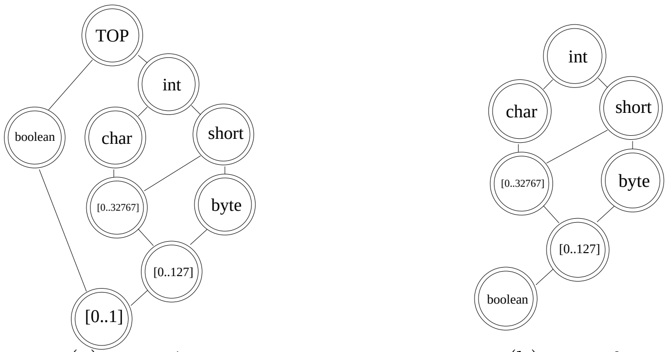
- Constraints collection.
- Merging connected components. (This may fail).
- Merging single relations by aplying the following rules until a fixed point is reached:
  - Replacing all multiple child dependencies between a single soft node and multiple hard nodes by a dependency on the *least common ancestor* type.
  - Replacing all multiple parent dependencies between a single soft node and multiple hard nodes by a dependency on the *greatest common descendent* type.
  - Merging a soft node with a single parent or single child hard node representing either `boolean, byte, short, char` or, `int`.

If this stage fails to deliver a solution (remaining soft node, conflicting parent or child constraints), then a second stage is performed using the type hierarchy in Figure 12(b) and the following steps:

- *Definition constraints* collection.
- Merging connected components. (This always succeeds).
- Merging single relation by aplying the following rules until a fixed point is reached:
  - Replacing all multiple child dependencies between a single soft node and multiple hard nodes by a dependency on the *least common ancestor* type.
  - Merging a soft node with a single child hard node.

This will always deliver a solution. In the final type assignment, [0..127] is replaced by `byte`, and [0..32767] is replaced by `char`. Finally, *use constraints* are verified and type casts are added as required.

The second stage might introduce *narrowing* type casts, and thus possibly change the semantics of the program. However, this would only happen when programs have dubious semantics to begin with. In our experiments, we have not discovered a case where stage 2 was needed.



(a) stage 1 (b) stage 2

Fig. 12. Integer type hierarchy

7 Experimental Results

The typing algorithm presented in this paper has been implemented in the Soot framework[21]. The typing algorithm accepts untyped Jimple as input, and outputs typed Jimple. Typed Jimple is used by subsequent analyses including class hierarchy analysis, pointer analysis and a Jimple to Java decompiler.

In this section we present the results of two set of experiments done using our implementation. The first set of experiments was performed to test the robustness of the typing algorithm as well as to gather empirical data about the complexity of type constraints in programs compiled from various languages. In the second experiment, the inferred types of Jimple were used to improve Class Hierarchy Analysis.

7.1 Typing Java bytecode

We have applied our typing algorithm on class files produced by compilers of five different languages: Java[10], Eiffel[20], Ada[23], Scheme[5] and ML[14]. Table 1 shows a selection of our results to show the general trends. The benchmarks are as follows: `javac` is the Sun's javac compiler, `jdk1.1` is everything in Sun's java class library for jdk1.1, `kalman` is a numeric Ada benchmark, `compile_to_c` is the SmallEiffel compiler (version 0.79), `lexgen` is a lexer generator used in the Standard ML of New Jersey benchmark suite, and `boyer` is one of the Gabriel Scheme benchmarks.

Language	Benchmark	# methods	conn. comp.	single cons.	stage 2	stage 3
java:	`javac`	1179	383	796	3	0
java:	`jdk1.1`	5060	2818	2228	14	0
ada:	`kalman`	735	463	262	10	0
eiffel:	`compile_to_c`	7521	1562	5959	0	0
ml:	`lexgen`	209	140	69	0	0
scheme:	`boyer`	2255	820	1433	2	0

Table 1. Required steps

The # *methods* column gives the total number of methods in the benchmark. The next two columns give the number of methods that could be typed using various steps of stage 1. The *conn. comp.* column counts the number of methods that could be completely typed by finding connected components, while the *single cons.* column counts the number of methods that needed both connected components and the removal of single constraints. The *stage 2* column counts the number of methods that required the stage 2. The *stage 3* column counts the number of methods that needed stage 3. It is interesting to note that a significant number of methods were typed using only connected components, and none of the 16959 methods required insertion of type casts (stage 3).

7.2 Improving class hierarchy analysis

One of the motivations for producing typed Jimple was for use in compiler analyzes and optimizations, and our second experiment illustrates one such use. In this experiment we measured the gains in the precision of the conservative call graph built using *class hierarchy analysis* (CHA)[4, 7, 8]. The basic idea is that for each virtual call of the form $o.f(a_1, a_2, \ldots, a_n)$, one needs to determine all possible methods f that could be called given the receiver o and a given class hierarchy. If the call graph is built using untyped Jimple, then type information provided by a method signature must be used to estimate the type of the receiver o. This type is correct, but may be too general and thus CHA may be too conservative about the destination of a virtual (or interface) call. If the call graph is built using typed Jimple, then each receiver has an inferred type provided by our algorithm, and this is often a tighter type than the type in the signature. This improved type information reduces the number of possible destinations for the call, and provides a better call graph on which further analysis can be made.

source language	program name	call-graph edges untyped Jimple (#)	call-graph edges typed Jimple (#)	Reduction (%)
java:	`jack`	10583	10228	3
java:	`javac`	26320	23625	10
java:	`jimple`	51350	33464	35
ada:	`rudstone`	8151	7806	4
eiffel:	`illness`	3966	3778	5
ml:	`nucleic`	5009	4820	4

Table 2. Call graph reduction

Table 2 shows the number of call-graph edges for both untyped and typed Jimple, and the percent reduction due to using typed Jimple. Again we present a selection of benchmarks from a variety of compilers. Note that the very object-oriented benchmarks like `javac` (10%) and `jimple` (35%) show significant reductions if the static type of the receiver is known. This means that subsequent analyses, including run-time type analysis, will start with a significantly better approximation of the call graph. The other benchmarks show a reduction in the 3% to 5% range, which is not as significant. This is mostly due to the fact that these benchmarks have a much simpler call graph to begin with, and so there is not much room for improvement.

These results serve to illustrate one benefit of typed Jimple. In fact the types are useful for a wide variety of other analyses including: (1) finding when an invokeinterface can be replaced by an invokevirtual call (i.e. when the inferred type of the receiver is a class, but the instruction is an invokeinterface), (2) deciding when a method can be safely inlined without violating access rules, (3) giving types to variables in decompiled code, and (4) as a basis for grouping variables by type (i.e. a coarse grain run-time type analysis or side-effect analysis can group variables by declared type).

8 Related Work

Related work has been done in the fields of type inference, typed assembly languages, and decompilation.

This work is a refinement of a preliminary report by Gagnon and Hendren[9]. In our preliminary work we proposed an exponential algorithm to solve difficult cases, whereas in this work we avoid the exponential case by applying program transformations, and we introduce the 3-stage approach. Further, this paper addresses the problem of assigning different integer types.

In [12], Knoblock and Rehof present a superficially similar algorithm to type Java bytecode. Their approach is different on many aspects. Their algorithm only works with programs in SSA form. It consists of adding new types and changing the interface hierarchy so that every two interfaces have a LUB and a SUP in the resulting type lattice. Changing the type hierarchy has unfortunate consequences: decompiled programs expose a type hierarchy that differs from the original program, the globality of such a change makes this algorithm useless in a dynamic code optimizers like HotSpot[22]. Our algorithm, on the other hand, works with any 3-address code representation and has no global side effects. It is thus suitable for use in a dynamic enviroment.

Type inference is a well known problem. There has been considerable work on type inference for modern object-oriented languages. Palsberg and Schwartzbach introduced the *basic type inference algorithm* for Object-Oriented languages [17]. Subsequent papers on the subject extend and improve this initial algorithm [18, 1, 2]. These algorithms infer dynamic types, i.e. they describe the set of possible types that can occur at runtime. Further, most techniques need to consider the whole program.

As we emphasized in the introduction, our type problem is different in that we infer static types. Further, we have a very particular property of having some type information from the bytecode, including the types of methods. This means that our type inference can be intra-procedural, and just consider one method body at a time.

Work has been done by Morrisett et al.[16] on stack-based typed assembly language. This work differs in that their typed assembly language is directly produced from a higher level language. Their work emphasizes the importance of having type information to perform aggressive optimizations. We agree that types are important for optimization, and this is one good reason we need our type inference.

Our technique is related to the type inference performed by Java decompilers[15, 11, 27, 3] and other Java compilers that convert from bytecode to C, or other intermediate representations. Proebsting and Watterson have written a paper[19] on decompilation in Java. Their paper is mainly focused on reconstruction high-level control statements from primitive goto branches. In their text, they wrongfully dismiss the type inference problem as being solvable by well known techniques similar to the Java verifier's algorithm. As we have shown in this paper, the problem is NP-Hard in general, and some bytecode programs require program transformations in order to be typeable statically.

9 Conclusion

In this paper we presented a static type inference algorithm for typing Java bytecode. We based our methods on a 3-address representation of Java bytecode called Jimple. In effect, we perform the translation of untyped 3-address code to typed 3-address code, where all local variables have been assigned a static type.

We have presented a constraint system that can be used to represent the type inference problem. Using this representation, we developed a simple, fast and effective multi-stage algorithm that was shown to handle all methods in a set of programs (and libraries) produced from five different source languages. We emphasized the difference between *well behaved* bytecode as defined by the Java verifier, and *well typed* bytecode, as required by a static typing algorithm. Our experimental results show that this efficient analysis can significantly improve the results of further analyzes like Class Hierarchy Analysis.

Acknowledgments

We thank Raja Vallée-Rai and other Sable research group members for their work on developing Jimple and the Soot framework.

References

1. Ole Agesen. Constraint-based type inference and parametric polymorphism. In Baudouin Le Charlier, editor, *SAS'94—Proceedings of the First International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 78–100. Springer, September 1994.
2. Ole Agesen. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26, Åarhus, Denmark, August 1995. Springer.
3. Ahpah Software Inc. http://zeus.he.net/~pah/products.html.
4. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31 of *ACM SIGPLAN Notices*, pages 324–341, New York, October 1996. ACM Press.
5. Per Bothner. Kawa - compiling dynamic languages to the Java VM, 1998.

6. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press; McGraw-Hill Book, Cambridge New York, 1990.
7. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Åarhus, Denmark, August 1995. Springer.
8. Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, La Jolla, California, June 1995.
9. Etienne M. Gagnon and Laurie J. Hendren. Intra-procedural inference of static types for java bytecode. Technical Report Sable 1998-5, McGill University, Montreal, Canada, October 1998. http://www.sable.mcgill.ca/publications/.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1997.
11. Innovative Software. http://world.isg.de.
12. T. Knoblock and J. Rehof. Type elaboration and subtype completion for Java bytecode. In *Proceedings 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*, pages 228–242, January 2000.
13. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, Jannuary 1997.
14. MLJ. http://research.persimmon.co.uk/mlj/.
15. Mocha. http://www.brouhaha.com/~eric/computers/mocha.html.
16. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Lecture Notes in Computer Science*, 1473:28–52, 1998.
17. Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.
18. J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.
19. Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (does bytecode reveal source?). In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 185–197, Berkeley, CA, USA, June 1997. USENIX.
20. Small Eiffel. http://SmallEiffel.loria.fr/.
21. Soot. http://www.sable.mcgill.ca/soot/.
22. Sun Microsystems Inc. http://java.sun.com/products/hotspot/.
23. Tucker Taft. Programming the Internet in Ada 95. In Alfred Strohmeier, editor, *Reliable software technologies, Ada-Europe '96: 1996 Ada-Europe International Conference on Reliable Software Technologies, Montreux, Switzerland, June 10–14, 1996: proceedings*, volume 1088, pages 1–16, 1996.
24. Jerzy Tiuryn. Subtype inequalities. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315, Santa Cruz, California, June 1992. IEEE Computer Society Press.
25. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON '99*, 1999.
26. Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode using the Soot framework: It is feasible? In David Watt, editor, *CC2000—International Conference on Compiler Construction*, pages 18–34, Berlin, Germany, March 2000.
27. WingSoft Corporation. http://www.wingsoft.com/wingdis.shtml.