McGill University
School of Computer Science
Sable Research Group

Intra-procedural Inference of Static Types for Java Bytecode [1]

Sable Technical Report No. 1999-1

Etienne Gagnon
Laurie Hendren

March 29, 1999

www.sable.mcgill.ca

[1]This work has been supported in part by FCAR and NSERC.

Abstract

Although Java bytecode has a significant amount of type information embedded in it, there are no explicit types for local variables. However, knowing types for local variables is very useful for both program optimization and decompilation. In this paper, we present a 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 are able to construct a sound and efficient algorithm for inferring the type of most bytecode found in practice. Further, for bytecode that cannot be typed, we suggest transformations that make it typeable.

Using this same constraint system, we show that this type inference problem is NP-hard in general. However, this hard case does not seem to arise in practice and our experimental results show that all of the 17,000 methods used in our tests were successfully and efficiently typed by our algorithm.

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 [18], ML [1], Scheme [7], and Eiffel [2].

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 2 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/optimization, the expression stack complicates both the analyzes 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. 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.[2]

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 analyzes. One example, improving the quality of the call graph, is presented in section 7.2.

[2]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.

In this paper, we address the problem of inferring a *static type* for each variable in a three-address representation of bytecode called Jimple. Jimple is part of the Soot compiler framework that is used for both compiler optimizations and decompilation.

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 collapsing nodes together. If nodes can be collapsed so that each collapsed node contains exactly one hard node, then we have found a solution to the typing problem. Figure 1(e) shows one possible collapsing 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 would prefer a typing that gives more specific types since this will help more in subsequent analyzes.

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, in Appendix A we build the necessary framework needed to show that the type inference problem is NP-complete. However, as our experimental results will show, in practice, a simple algorithm is able to solve most problems in polynomial time. Further, we show program transformations that allows us to type all of the hard cases using the simple algorithm.

It is important to contrast this work, where we find a *static* type consistent with *all* uses of a variable, with other type inference analyzes 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 [16, 12, 4, 5]. We will call such algorithms *run-time type analyzes* to distinguish them from our *static-type analysis*. For our example program in Figure 1(a), run-time type analyzes 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 analyzes, including a run-time type analyzes we have built on top of typed Jimple.

Our paper is structured as follows. In section 2 we show some examples to demonstrate why this typing problem is difficult. In section 3 we present the Jimple representation. In section 4, we define the general static type inference problem, and give the main algorithm for programs without arrays and in section as well as an overview of the complexity of the algorithm. In section 5 we present extensions to our algorithm to handle arrays, and in Section 6 we show how to deal with the other difficult cases by introducing copy statements. Section 7 contains our experimental results. Finally, we briefly review related work in section 8 and present our conclusions in section 9.

```
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);
}
```

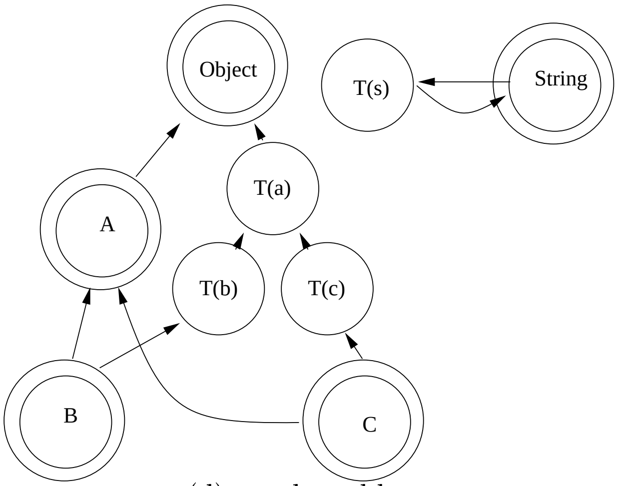
(a) untyped method

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

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

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

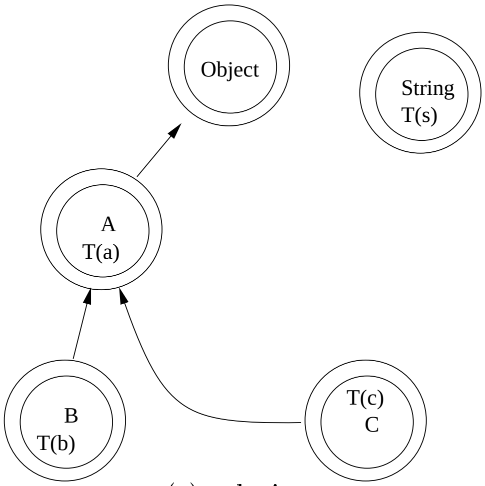
(b) class hierarchy

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

Figure 1: Simple example of static typing

2 Challenges of Typing

The static typing problem looks quite simple at first, but there are several subtle points that make the problem quite interesting. In this section we illustrate two difficulties and we demonstrate the differences between our typing problem and the typing approximation done by the Java verifier. A third difficulty, that of correctly typing arrays, is explained in Section 5.

2.1 Declared variable types versus types at program points

Part of the static Java verifier is a flow analysis that estimates, at each program point, the type of 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 the example in Figure 2(a), where the class hierarchy is defined as in Figure 2(b). The bytecode verifier would infer that the type of `a` at program point `s1` is `A` and at program point `s2` is is `B`. At the control flow merge point just before `s3`, the verifier takes the *least common ancestor class* or LCAC of the types on each branch, in this case $type(a) = LCAC(A, B) = Object$. Thus, the type of `a` at program point `s3` would be `Object`. Since all of the types are correct with respect to the verifier, this method would verify.

Now consider our problem of finding one static type for `a`. In this case there is no solution, even though the bytecode is verifiable. If we chose type `A`, then the type at statement `s2` is wrong, if we chose type `B`, the type at statement `s1` is wrong, and if we chose type `Object`, the types at both statements `s1` and `s2` are wrong. In fact, one can not write a Java program like this Jimple program and give a correct static type to `a`. However, remember that our Jimple code comes from bytecode produced from any compiler or bytecode optimizer, and so this situation may (and does) occur. In order to give static types some transformation must be done on the program. Several decompilers chose the option of giving `a` the type `Object` and then introducing casts at statements `s1` and `s2`. This is not necessarily the best solution because a static type check is replaced by a dynamic check. In Section 6 we show a different solution where we introduce extra copies of variables, and each copy can be given a correct static type.

```
void m() {
     <unknown> a;
     if (...)
       { a = new A();
 s1:     a.f(); // invokevirtual A.f()
       }
     else
       { a = new B();
 s2:     a.g(); // invokevirtual B.g()
       }
 s3: a.toString();
       // invokevirtual Object.toString()
 }
```

(a) untyped method

```
class Object {
   public String toString() { .... }
   ... }

class A extends Object {
   public void f() { .... }
   ... }

class B extends Object {
  public void g() { .... }
  ... }
```

(b) hierarchy

Figure 2: Different types needed at different program points

2.2 Type problems due to interfaces (multiple inheritance)

The verifier's simple strategy of taking the LCAC to resolve types coming from different branches of control flow works very nicely for single inheritance, but it breaks down in the presence of interfaces which give a restricted form of multiple inheritance. In the Java verifier this problem is dealt with by using the simple scheme for the static analysis, and actually checking the types for invokeinterface calls as part of the run-time verification. In our static analysis we wish to give correct static types for interfaces, and thus eliminate the need for dynamic checks.

Consider the example program in Figure 3(a). The Java verifier would infer that at program point `s1` `a` has interface type `IC`, and at program point `s2` `a` has interface type `ID`. At the merge point just before `s3`, the simple solution of taking the LCAC would give the type of `a` as `Object`. Thus, the invokeinterface calls at `s3` and `s4` cannot be statically checked, and a dynamic verification is required to make sure that method `f` is defined for the call at `s3` and method `g` is defined for the call at `s4`.

If we statically type this example using hierarchy I, we could find that `IMiddle` is a type that satisfies all constraints, and thus the dynamic checks are not needed. As we show later, finding such an interface may be expensive, since there is not necessarily a lowest common ancestor, but potentially many, unrelated, common ancestors due to multiple inheritance. Further, if `IMiddle` does not exist, as in Hierarchy II, then there is no static type satisfying all constraints. In both cases, introducing extra copies provides a statically typeable solution. We discuss this further in Section 6.

3 The Jimple Representation

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, the Sable Research Group developed *Jimple* [19], a 3-address-code representation of bytecode, where all stack-based operations are transformed into local variable based operations. This is made possible by the restrictions 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 Jimple 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[3].

For example:

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

[3] In reality, the stack analysis, the introduction of new local variables, and the transformation are not as straightforward 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 [19].

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

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

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

  void test()
  { <untyped> a;

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

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

(a) untyped program

```
[Hierarchy I]
Interface IA { void a(){} }
Interface IB { void b(){} }
Interface IMiddle extends IA, IB {}
Interface IC extends IMiddle {}
Interface ID extends IMiddle {}


[Hierarchy II]
Interface IA { void a(); }
Interface IB { void b(); }
Interface IC extends IA, IB {}
Interface ID extends IA, IB {}
```

(b) hierarchy I and II

Figure 3: Typing interfaces

```
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
```

The Jimple representation retains 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, the untyped version of Jimple does not have types for local variables. 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 change 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. Also note that this form of Jimple looks overly long, with many spurious copy statements. In the real Jimple the code is cleaned up using standard techniques for copy propagation and elimination.

4 Typing 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.

The type inference algorithm consists of applying successive transformations on the constraint graph in order to find a solution as efficiently as possible. But, as shown in Appendix A, in general, finding a static-type assignment for a Jimple program is NP-Hard. Furthermore, as we have seen in section 2, not all verifiable bytecode programs can be statically typed. For this reason, we will also present program transformations that will allow us to build semantically equivalent programs that can be efficiently typed.

We initially restrict our problem to programs that do not include arrays, nor array operations. This allows us to illustrate the constraint system. Solving this restricted problem consists of building a directed-graph of program constraints, finding and merging connected components of this graph, removing transitive constraints, merging single constraints, and optionally, if no solution is found yet, doing a *worst-case* exponential-time exhaustive search. In this restricted solution, we halt the algorithm if any untypeable problem is found. Next, we discuss the theoretical foundations of our algorithm, sketching a soundness proof and complexity analysis for each step.

Once we have established the initial constraint system, we extend it with extra notation for array constraints. We then show how we can 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.

Finally, we present two program transformations that allow to type untypeable Jimple programs, and optionally avoid exhaustive type search when a complex problem is found. One transformation introduces explicit type casts in the original program, the other adds copy statements.

4.2 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[4].

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.
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. The rules of *assignment compatibility* are explained in [13] and [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 $T(a)$ to $T(b)$ (if not already present). This is shown in figure4.

The assignment of the following *binary expression* to local variable a, $[a = b + 3]$, generates the following constraints: $T(a) \leftarrow T(b)$, $T(a) \leftarrow int$, and $T(b) \leftarrow int$.

Our last and most complicated case is a method invocation, $[a = b.equals(c)]$, or in full:

[4]In this paper, we do only infer the primitive types defined in [13]. In other words, *boolean*, *byte*, *short*, and *char* are treated as *int*.

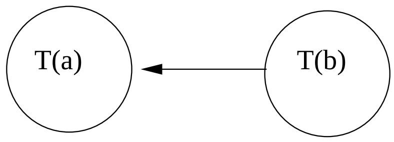


Figure 4: b assigned to a

```
a = virtualinvoke b.[boolean java.lang.Object.equals(java.lang.Object)] (c)
```

A method invocation carries much type information. So we get all the following constraints, each involving a hard node:

- $T(a) \leftarrow int$, because the return type of *equals* is *boolean* (we have a single integer type).
- $java.lang.Object \leftarrow T(b)$, from the declaring class of *equals*.
- $java.lang.Object \leftarrow T(c)$, from the argument type in the method signature.

As shown in example 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 if a node needs more than one associated type, the type inference algorithm should halt and report that no solution exists.

4.3 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.

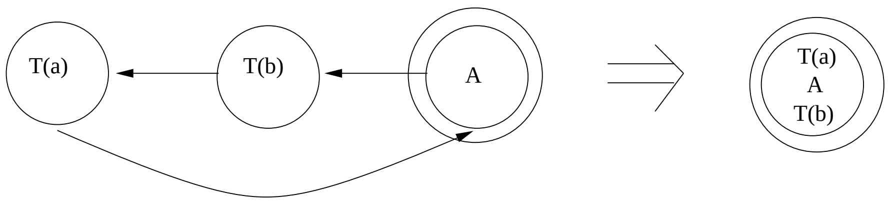


Figure 5: Merging Connected Components

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 are be merged together to the same hard node, in any solution to the constraint graph.

We can divide connected components into three kinds. Firstly, 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[5]. Secondly, 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.

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

If any constraint can't be satisfied, the algorithm halts. Thirdly, it happens that a connected component has two or more hard nodes (as would happen with Figure 2. When this occurs, the algorithm halts.

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.

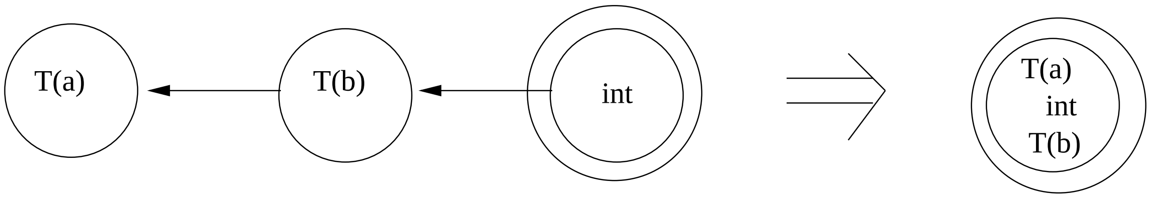


Figure 6: Merging Primitive Types

4.4 Transitive Constraints

Once the connected components are removed from the constraint graph, we are left with a directed-acyclic-graph (DAG). Our next transformation consist of removing spurious constraints (edges) from this DAG. By spurious constraint, we mean a constraint which is already expressed otherwise in the graph. More specifically, we simply eliminate the transitive constraints of 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). This is shown in figure 7.

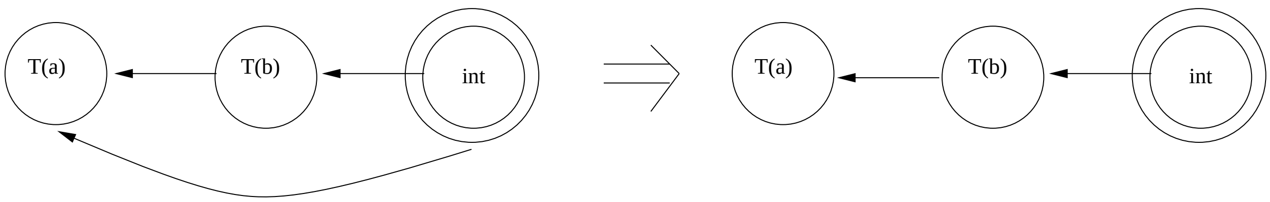


Figure 7: Merging Transitive Constraints

4.5 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 in four steps:

1. Merge all 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 all single soft parent constraints: Merge x with y when x is a soft node with a single parent constraint to another *soft node* y.

3. Merge with least common ancestors: This is a special case. When x is a soft node that only has constraints to hard nodes representing *class types*, and x has a single parent constraint, then we merge x with the hard node representing the least common ancestor of all children class types. This is similar to the static part of the Java verifier algorithm, but it differs in that it is not applied if any involved hard node represents an *interface type*.

4. Merge all remaining single parent constraints: Merge x with y when x is a soft node with a single parent constraint to another node y.

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 descendent 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.

An example of merging single constraints is shown in Figure 8.

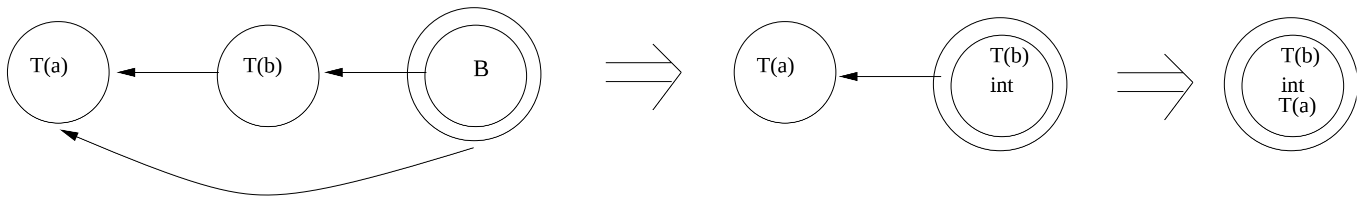


Figure 8: Merging Single Constraints

It is a little trickier to see why this merging is justified. As an example, consider merging a soft node with a single parent. The argument is that if a type inference solution exists where the merged soft node has a different type than the *merged-to* parent node, then we can build another type inference solution by simply replacing the type assignment of the soft node by the type of the *merged-to* parent node. This is because all child constraints of the soft node are indirect child constraints of the parent, so they must be met by the type assignment of the parent. In addition, since the soft node has a single parent constraint, all its indirect parent constraints are either direct or indirect parent constraints of the parent node, and must thus be met by the type assignment of the parent. Finally, $x \leftarrow x$ is always true, so the single parent constraint is satisfied. So all constraints of the soft node are still satisfied using this new assignment. Other cases can be argued similarly.

4.6 Exhaustive Search

Once we have removed all single constraints from the graph, if there are any soft nodes left, then it must have at least two parent and two child constraints. In Appendix A we show that solving this problem is NP-Hard. So, we pick the obvious, exponential worst-case time, exhaustive search algorithm. We successively assign to each soft node all possible types in the part of the type hierarchy delimited by explicitly referenced types in the method body, until we find a solution (if any). This can theoretically be expensive, but as our experimental results show, this case does not happen in practice. Further, the type hierarchy required to cause a *worst-case* behavior involves a very complex multiple inheritance interface hierarchy which isn't likely to ever happen in real programs. See Appendix A for more details.

4.7 Complexity

In this section, we discuss the complexity of the algorithm presented in the previous section.

The input to the typing algorithm can be broken into two parts: (1) the method body, and (2) the type hierarchy. It is fairly obvious that the size of the constraint graph (nodes + edges) is linear in the number of statements in the method body, as each statement can cause the addition of a small constant number of nodes and constraints. The size of the type hierarchy, can have size proportional to the whole program. On the other hand, type hierarchy information can be computed once and stored, then reused when the type inference algorithm is invoked on different methods of the program. In a practical context, as in a compiler, this information is usually already computed independently from the typing algorithm.

If we ignore the type hierarchy size factor, all steps of the constrained type inference algorithm can be performed in polynomial time (in the size of the method). In fact, most parts can be done in linear time in the size of the method body (i.e. number of statements). Constraint collection is obviously linear. A well known algorithm finds *strongly connected components* in a directed graph [8] in linear time in the size of the graph, thus in the size of the method body. All other steps in the algorithm, such as the transitive constraint collection, and all merging steps can be done in $O(n^2)$. Only the exhaustive search can break the polynomial complexity. This latest step can be kept optional using program transformation techniques to find a type solution in polynomial time.

5 Array Constraints

The main idea, for handling arrays, is to introduce a new kind of constraint in the constraint graph to represent array constraints. An *array constraint* embodies the relation between the type of an array and the type of its element. We write it as $a \mapsto b$, to indicate a is a array type which element type is b.

This simple addition allows us to collect constraints for all Jimple statements. For example:

```
a[d] = b;
c = a;
```

generates the following constraints: $a \mapsto b, d \leftarrow int$, and $c \leftarrow a$.

We then compute maximal array depths. The *array depth* is the number of indirection required to get a non-array type. For example, $A[][]$ has array depth 2. All hard nodes have constant array depth[6]. Assignment compatibility rules restrict assignments from an array depth to the same or to a smaller array depth. For example: $B[]$ can be assigned to $Object$, but C cannot be assigned to $D[]$.

Using this simple knowledge, we start from hard nodes (using a work list algorithm), and we follow (1) parent constraints, modifying the parent depth to insure that the maximal depth of the parent is less or equal to the depth of each of its children, and (2) array constraints, assigning to the element type, the array depth minus one[7].

The next step is to propagate all constraints on arrays down from non-depth 0 nodes to depth 0 nodes using the following rules:

- Propagate a constraint between two nodes of equal depth to a constraint between their depth 0 element types.
- Change a constraint between two nodes of different depth to a constraint between the depth 0 element type of lowest depth node and *java.lang.Cloneable* (and *java.io.Serializable*).

This is illustrated in figure 9.

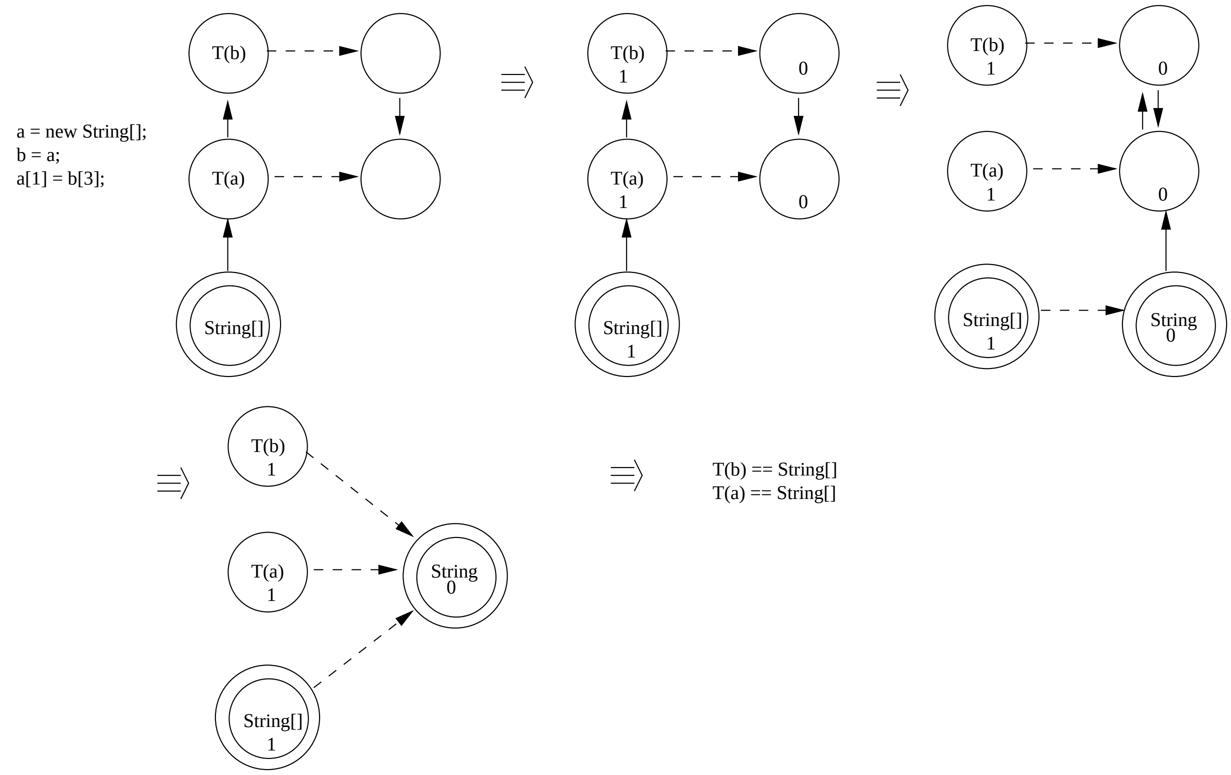


Figure 9: Solving Array Constraints

[6]A special case in *null*, which has infinite array depth, since it can extend all array types of any depth.
[7]Once the fixed point is reached, *null* becomes a depth 0 type.

The resulting constraint graph allows us to use the constrained algorithms to solve the constraints on the graph consisting of only depth 0 nodes (and constraints between them).

Once a solution is found on depth 0, it is propagated back to array depths intuitively. For example, if $x \mapsto y$, and $y = A$, then $x = A[]$

The rules for array constraint propagation are directly justified by the assignment compatibility of Java array types[8].

In order to handle correctly the case of primitive array types ($boolean[]...[]$, $short[]...[]$, $char[]...[]$, $byte[]...[]$), we merge these hard nodes with all their same-depth neighbors[9].

6 Program Transformations

In order to type untypeable programs, or reduce the worst-case time complexity of the type inference algorithm, two program transformations can be applied, when a difficult case is found.

6.1 Type Casts

The first technique, commonly in use in existing decompilers, is the addition of type casts.

Using type casts, we can render the example of Figure 3 typeable, by inserting a type cast as Follows:

```
s3:   ((IA) a).f();
```

This technique has the advantage of being simple and effective. But it has the drawback of adding a *checkcast* instruction which can be quite difficult to remove later, if bytecode is generated out of an optimized Jimple program. This can have the unfortunate consequence of slowing down the program, as the checkcast instruction can be costly on some java interpreters.

6.2 Copy Statements

The second technique consists of introducing copy statements. We can make 2 typeable as follows:

```
void m() {
     <unknown> a;
     <unknown> b;
     <unknown> c;
     if (...)
       { a = new A();
 s1:     a.f(); // invokevirtual A.f()
         c = a; // Extra copy
       }
     else
       { b = new B();
 s2:     b.g(); // invokevirtual B.g()
         c = b; // Extra copy
       }
```

[8] $A[]$ is assignable to $B[]$ if and only of A is assignable to B.

[9] This is necessary because the depth 0 for all these types is int

```
 s3: c.toString();
       // invokevirtual Object.toString()
 }
```

A simple version of copying consists of introducing copies following *new* statements to take care of the common case of the creation of instances on two branches of a Java if statement, as seen in Figure 2. This version cannot handle the example in figure 3.

The advantage of copy statements is that well known techniques exist to get rid of extra copy statements (e.g. copy propagation). On the other hand the complete copy algorithms consists of splitting all neighbour variables according to def-use chains. But, this complete version tends to introduce too many copies, as shown here.

```
void m() {
     <unknown> a;
     <unknown> b;
     <unknown> c;
     <unknown> t1;
     <unknown> t2;
     if (...)
       { t1 = new A();
         a = t1;
         c = t1;
 s1:     a.f(); // invokevirtual A.f()
       }
     else
       { t2 = new B();
         b = t2;
         c = t2;
 s2:     b.g(); // invokevirtual B.g()
       }
 s3: c.toString();
       // invokevirtual Object.toString()
 }
```

7 Experimental Results

The typing algorithm presented in this paper has been implemented in the Soot framework[3]. The typing algorithm accepts untyped Jimple as input, and outputs typed Jimple. Typed Jimple is used by subsequent analyzes 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[2], Ada[18], Scheme[7] and ML[1]. 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	# transf.	conn. comp.	single cons.	exhaust.
java:	`javac`	1179	3	383	796	0
java:	`jdk1.1`	5060	14	2832	2228	0
ada:	`kalman`	735	10	473	262	0
eiffel:	`compile_to_c`	7521	0	1558	5959	0
ml:	`lexgen`	209	0	140	69	0
scheme:	`boyer`	2255	2	820	1433	0

Table 1: Required steps

The *transf.* column counts the number of methods that required the simple splitting algorithm transformation before typing. Not shown in the table is the fact that the simple splitting algorithm failed only once to render a method typeable in the Eiffel benchmark. In this case, the complete splitting algorithm was used. The next three columns give the number of methods that could be typed using various steps of the algorithm. 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 *exhaust.* column counts the number of methods that needed the exhaustive search. It is interesting to note that a significant number of methods are typed using only connected components, and none of the 16959 methods required an exhaustive search.

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). 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 large 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 may be 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 analyzes can be made.

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 analyzes, 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

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

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 analyzes 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.

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 [16]. Subsequent papers on the subject extend and improve this initial algorithm [12, 4, 5]. 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.[15] 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 techniques are more related to the kind of type inference performed by Java decompilers[14, 11, 20, 6] and other Java compilers that convert from bytecode to C, or other intermediate representations. While many commercial Java decompilers can be found, literature on Java decompilation is sparse. Proebsting and Watterson have written a paper[17] 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 verifier's static algorithm is incomplete as it requires dynamic checks. Further, the problem is NP-Hard in general, and program transformations are required in order to be able to type any bytecode program.

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 call Jimple. In effect, we perform the translation of untyped Jimple to typed Jimple, 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 simple, fast and effective algorithms that were 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 practically cheap analysis can significantly improve the results of further analyzes like Class Hierarchy Analysis.

10 Acknowledgments

We offer our special thanks to Michael Schwartzbach for providing us with the basic construction we use in the NP-complete proof. Also, thanks to Raja Vallée-Rai for his work on developing the Soot and Jimple framework.

References

[1] URL: http://research.persimmon.co.uk/mlj/.

[2] URL: http://SmallEiffel.loria.fr/.

[3] URL: http://www.sable.mcgill.ca/soot/.

[4] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *First International Static Analysis Symposium*, September 1994.

[5] Ole Agesen. The cartesian product algorithm : Simple and precise type inference of parametric polymorphism. In *ECOOP '95*, Aarhus, Denmark, August 1995.

[6] Ahpah Software Inc. `http://zeus.he.net/~pah/products.html`.

[7] Per Bothner. Kawa - compiling dynamic languages to the java vm, 1998. URL: http://sourceware.cygnus.com/kawa/papers/.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press; McGraw-Hill Book, Cambridge, ; New York,, 1990.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freemann and Co., New York,, 1979.

[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] J.Plevyak and A.Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94*, pages 324 – 340, October 1994.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, jan 1997.

[14] Mocha. `http://www.brouhaha.com/~eric/computers/mocha.html`.

[15] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Lecture Notes in Computer Science*, 1473:28–??, 1998.

[16] Jens Palsberg and Michael I.Schwartzbach. Object oriented type inference. In *OOPSLA '91*, Phoenix, Arizona, October 1991.

[17] 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, 1997. USENIX. The authors demonstrate that, because of the strongly-type nature of the Java Virtual Machine, it is possible to reverse-engineer Java byte codes back to quite readable correct Java source code, making it difficult to conceal Java source code by distributing only compiled byte codes. This has significant ramifications for commercial software development in Java.

[18] 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 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag. Describes the Intermetrics Ada 95 compiler, which generates Java Virtual Machine bytecodes directly.

[19] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, McGill University, July 1998.

[20] WingSoft Corporation. `http://www.wingsoft.com/wingdis.shtml`.

A NP-Hard Proof elements

In this appendix, we show the main construction that can be used to prove that solving the restricted constraint graph problem is NP-complete. The proof consists in reducing the well known 3-SAT problem [9] to our graph problem.

It is easy to see that our problem is a *yes/no* problem, and that a certificate could be checked in polynomial time. In the remaining of this section, we will only provide the necessary material to carry the reduction through, and leave to the interested reader the task of completing the proof.

First, we should note that to completely specify a constraint graph problem (CG), we need two graphs: (1) the class hierarchy and (2) the constraint graph of soft and hard nodes.

Given an instance of 3-SAT, we build an instance of CG. We first build the class hierarchy.

- interface Top
- For each variable x_i,
  - interface x_i^{top}
  - interface x_i extends Top, x_i^{top}
  - interface x_i^{not} extends Top, x_i^{top}
  - interface x_i^{bottom} extends x_i, x_i^{not}

- For each clause $C^i = (c_1^i, c_2^i, c_3^i)$,
  - interface C_i^{top}
  - c_1^i is equal to x_j or x_j^{not} for some j. We introduce interface c_1^i extends *all interfaces* x_k *and* x_k^{not} *where* $k \neq j$ *and (*x_j *when* c_1^i *is equal to* x_j*, or* x_j^{not} *otherwise)*
  - interfaces c_2^i and c_3^i, are similar to c_1^i.
  - interface C_i^{bottom} extends c_1^i, c_2^i, c_3^i
- interface Bottom extends *all interfaces* c_k^j

Then we build the constraint graph.

- hard nodes: Top and Bottom
- For each variable x_i,
  - hard nodes: x_i^{top} and x_i^{bottom}
  - soft node: x_i
  - constraints: Top $\leftarrow x_i$, $x_i^{top} \leftarrow x_i$, $x_i \leftarrow x_i^{bottom}$,
- For each clause $C^i = (c_1^i, c_2^i, c_3^i)$,
  - hard nodes: C_i^{top} and C_i^{bottom}
  - soft node: c_i
  - c_1^i is equal to x_j or x_j^{not} for some j. We introduce the constraint:$x_j \leftarrow c_i$
  - similarly, constraints $x_k \leftarrow c_i$ and $x_m \leftarrow c_i$ are introduced
- constraints: $c_i \leftarrow$ Bottom (for all i)

The intuition to understand the construct is to note that the soft node x_i should resolve to either interface x_i or x_i^{not}. This gives us a truth assignment for each variable. Also, the soft node c_i should resolve to either interface c_1^i, c_2^i or c_3^i, which corresponds to a *true* value satisfying the clause.