Class Hierarchy Complementation: Soundly Completing a Partial Type Graph

George Balatsouras Yannis Smaragdakis

Department of Informatics
University of Athens, 15784, Greece
{gbalats,smaragd}@di.uoa.gr

Abstract

We present the problem of class hierarchy complementation: given a partially known hierarchy of classes together with subtyping constraints ("A has to be a transitive subtype of B") complete the hierarchy so that it satisfies all constraints. The problem has immediate practical application to the analysis of partial programs—e.g., it arises in the process of providing a sound handling of "phantom classes" in the Soot program analysis framework. We provide algorithms to solve the hierarchy complementation problem in the single inheritance and multiple inheritance settings. We also show that the problem in a language such as Java, with single inheritance but multiple subtyping and distinguished class vs. interface types, can be decomposed into separate single- and multiple-subtyping instances. We implement our algorithms in a tool, JPhantom, which complements partial Java bytecode programs so that the result is guaranteed to satisfy the Java verifier requirements. JPhantom is highly scalable and runs in mere seconds even for large input applications and complex constraints (with a maximum of 14s for a 19MB binary).

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; D.3.4 [*Programming Languages*]: Processors—Compilers; D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords type hierarchy; Java; single inheritance; multiple inheritance; JPhantom; bytecode engineering

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509530

1. Introduction

Whole-program static analysis is essential for clients that require high-precision and a deeper understanding of program behavior. Modern applications of program analysis, such as large scale refactoring tools [9], race and deadlock detectors [16], and security vulnerability detectors [11, 15], are virtually inconceivable without whole-program analysis.

For whole-program analysis to become truly practical, however, it needs to overcome several real-world challenges. One of the somewhat surprising real-world observations is that whole-program analysis requires the availability of much more than the "whole program". The analysis needs an overapproximation of what constitutes the program. Furthermore, this overapproximation is not merely what the analysis computes to be the "whole program" after it has completed executing. Instead, the overapproximation needs to be as conservative as required by any intermediate step of the analysis, which has not yet been able to tell, for instance, that some method is never called.

Consider the example of trying to analyze a program P that uses a third-party library L. Program P will likely only need small parts of L. However, other, entirely separate, parts of L may make use of a second library, L'. It is typically not possible to analyze P with a whole program analysis framework without also supplying the code not just for L but also for L', which is an unreasonable burden. In modern languages and runtime systems, L' is usually not necessary in order to either compile P or run it under any input. The problem is exacerbated in the current era of large-scale library reuse. In fact, it is often the case that the user is not even aware of the existence of L' until trying to analyze P.

Unsurprisingly, the issue has arisen before, in different guises. The FAQ document[1] of the well-known Soot framework for Java analysis [19, 20] contains the question:

> *How do I modify the code in order to enable soot to continue loading a class even if it doesn't find some of it[s] references? Can I create a dummy soot class so it can continue with the load? How?*

[1] http://www.sable.mcgill.ca/soot/faq.html

This frequently asked question does not lead to a solution. The answer provided is:

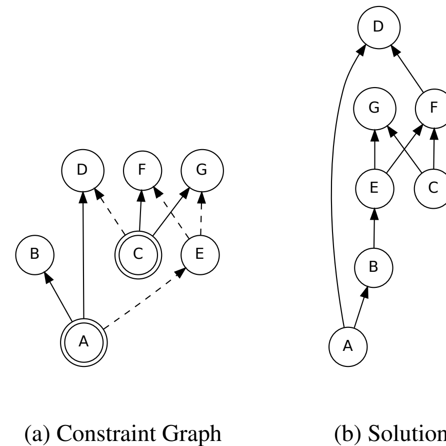
> *You can try -use-phantom-refs but often that does not work because not all analyses can cope with such references. The best way to cope with the problem is to find the missing code and provide it to Soot.*

The “phantom refs” facility of Soot, referenced in the above answer, attempts to model missing classes (*phantom classes*) by providing dummy implementations of their methods referenced in the program under analysis. However, there is no guarantee that the modeling is in any way sound, i.e., that it satisfies the well-formedness requirements that the rest of the program imposes on the phantom class.

Our research consists precisely of addressing the above need in full generality. *Given a set of Java class and interface definitions, in bytecode form, we compute a “program complement”, i.e., skeletal versions of any referenced missing classes and interfaces so that the combined result constitutes verifiable Java bytecode.* Our solution to this practical problem has two parts:

- A *program analysis* part, requiring analysis of bytecode and techniques similar to those employed by the Java verifier and Java decompilers. This analysis computes constraints involving the missing types. For instance, if a variable of a certain type C is direct-assigned to a variable of a type S, then C must be a subtype of S.
- An *algorithmic* part, solving a novel typing problem, which we call the *class hierarchy complementation*, or simply *hierarchy complementation*, problem. The problem consists of computing a type hierarchy that satisfies a set of subtyping constraints *without* changing the direct parents of known types.

The algorithmic part of our solution, i.e., solving the hierarchy complementation problem, constitutes the main novelty of our approach. The problem appears to be fundamental, and even of a certain interest in purely graph-theoretic terms. For a representative special case, consider an object-oriented language with multiple inheritance (or, equivalently, an interface-only hierarchy in Java or C#).[2] A partial hierarchy, augmented with constraints, can be represented as a graph, as shown in Figure 1a. The known part of the hierarchy is shown as double circles and solid edges. Unknown (i.e., missing) classes are shown as single circles. Dashed edges represent subtyping constraints, i.e., indirect subtyping relations that have to hold in the resulting hierarchy. In graph-theoretic terms, a dashed edge means that there is a path in the solution between the two endpoints. For instance, the dashed edge from C to D in Figure 1a means that the unknown part of the class hierarchy has a path from C to D. This path cannot be a direct edge from C to D, however: C is a known class, so the set of its supertypes is fixed.



(a) Constraint Graph (b) Solution

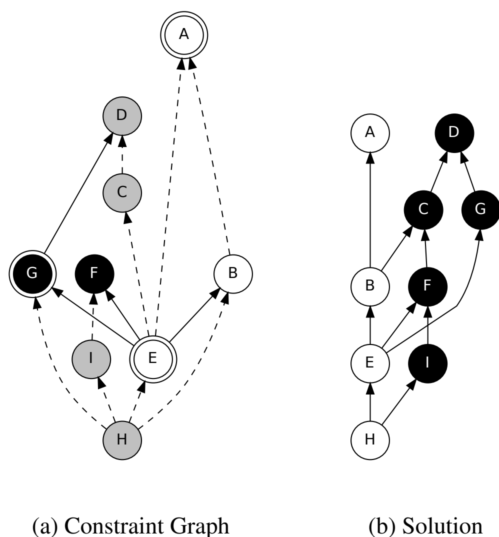
Figure 1: Example of constraints in a multiple inheritance setting. Double-circles signify known classes, single circles signify unknown classes. Solid edges (“known edges”) signify direct subtyping, dashed edges signify transitive subtyping.

In order to solve the above problem instance, we need to compute a directed acyclic graph (DAG) over the same nodes,[3] so that it preserves all known nodes and edges, and adds edges *only to unknown nodes* so that all dashed-edge constraints are satisfied. That is, the solution will not contain dashed edges (indirect subtyping relationships), but every dashed edge in the input will have a matching directed path in the solution graph. Figure 1b shows one such possible solution. As can be seen, solving the constraints (or determining that they are unsatisfiable) is not trivial. In this example, any solution has to include an edge from B to E, for reasons that are not immediately apparent. Accordingly, if we change the input of Figure 1a to include an edge from E to B, then the constraints are not satisfiable—any attempted solution introduces a cycle. The essence of the algorithmic difficulty of the problem (compared to, say, a simple topological sort) is that we cannot add extra direct parents to known classes A and C—any subtyping constraints over these types have to be satisfied via existing parent types. This corresponds directly to our high-level program requirement: we want to compute definitions for the missing types only, without changing existing code.

For a language with single inheritance, the problem is similar, with one difference: the solution needs to be a tree

[2] We avoid the terms “subclassing” or “inheritance” as synonyms for “direct subtyping” to prevent confusion with other connotations of these terms. In our context, we only care about the concept of subtyping, i.e., of a (monomorphic) type as a special case of another. Subtyping can be direct (e.g., when a Java class is declared to “extend” another or “implement” an interface) or indirect, i.e., transitive. We do, however, use the compound terms “single inheritance” and “multiple inheritance” as they are more common in the classification of languages than “single subtyping” and “multiple subtyping”.

[3] Inventing extra nodes does not contribute to a solution in this problem.



(a) Constraint Graph (b) Solution

Figure 2: Example of full-Java constraint graph. Double circles denote known classes/interfaces, whose outgoing edges in the solution are already determined (solid input edges). White nodes are classes, black nodes are interfaces, grey nodes are unknown types that are initially undetermined (i.e., the input does not explicitly identify them as classes or interfaces, although constraint reasoning may do so later).

instead of a DAG. (Of course, the input in Figure 1a already violates the tree property since it contains known nodes with multiple known parents.) We offer an algorithm that solves the problem by either detecting unsatisfiability or always ordering the nodes in a tree that respects all constraints.

The practical version of the hierarchy complementation problem is more complex. Mainstream OO languages often distinguish between classes and interfaces and only allow single direct subtyping among classes and multiple direct subtyping from a class/interface to an interface—a combination often called "single-inheritance, multiple subtyping". In this case, the graph representation of the problem is less intuitive. Consider Figure 2a that gives a problem instance. (A possible solution for these constraints is in Figure 2b, but is given purely for reference, as it is not central to our discussion.) There are now several node types: classes, interfaces (both known and unknown), as well as undetermined nodes. There are also more implicit constraints on them: classes can only have an edge to one other class, interfaces can only have edges to other interfaces. The latter constraint, for instance, forces D to be an interface and H to be a class. Thus, we see that the full version of the problem requires additional reasoning. We show that such reasoning can be performed as a pre-processing step. The problem can be subsequently broken up into two separate instances of the aforementioned single- and multiple-inheritance versions of hierarchy complementation.

In brief, the contributions of our work are as follows:

- We introduce a new typing problem, motivated by real-world needs for whole program analysis. To our knowledge, the hierarchy complementation problem has not been studied before, in any context.
- We produce algorithms that solve the problem in three different settings: single inheritance, multiple inheritance, and mixture of the two, as in Java or C#.
- We implement our algorithms in JPhantom: a practical tool for Java program complementation that addresses the soundness shortcomings of previous Java "phantom class" approaches. We show that JPhantom scales well and takes only a few seconds to process even large benchmarks with complex constraints—e.g., less than 6sec for a 3.2MB binary that induces more than 100 constraints.
- We discuss the problem of hierarchy complementation in more general settings. The simplicity of our approach is a result of only assuming (for the input) and satisfying (for the output) the fairly weak Java bytecode requirements. We show that the problem becomes harder at the level of the type system for the source language.

2. Motivation and Practical Setting

We next discuss the practical setting that gives rise to the hierarchy complementation problem.

Our interest in hierarchy complementation arose from efforts to complement existing Java bytecode in a way that satisfies the soundness guarantees of the Java verifier. Consider a small fragment of known Java bytecode and the constraints it induces over unknown types. (We present bytecode in a slightly condensed form, to make clear what method names or type names are referenced in every instruction.) In this code, classes `A` and `B` are available, while types `X`, `Y`, and `Z` are phantom, i.e., their definition is missing.

```
public void foo(X, Y)
0: aload_2     // load on stack 2nd argument (of type Y)
1: aload_1     // load on stack 1st argument (of type X)
2: invokevirtual X.bar:(LA;)LZ; //method 'Z bar(A)' in X
3: invokevirtual B.baz:()V;    //method 'void baz()' in B
 ...
```

The instructions of this fragment induce several constraints for our phantom types. For instance:

- `X` has to be a class (and not an interface) since it contains a method called via the `invokevirtual` bytecode instruction.
- `X` has to support a method `bar` accepting an argument of type `A` and returning a value of type `Z`.
- `Y` has to be a subtype of `A`, since an actual argument of declared type `Y` is passed to `bar`, which has a formal parameter of type `A`. This constraint also means that if `A` is known to be a class (and not an interface) then `Y` is also a class.

- `Z` has to be a subtype of `B`, since a method of `B` is invoked on an object of declared type `Z` (returned on top of the stack by the earlier invocation).

The goal of our JPhantom tool is to satisfy all such constraints and generate definitions of phantom types `X`, `Y`, and `Z` that are compatible with the bytecode that is available to the tool (i.e., exists in known classes). Compatibility with existing bytecode is defined as satisfying the requirements of the Java verifier, which concern type well-formedness.

Of these constraints, the hardest to satisfy are those involving subtyping. Constraints on members (e.g., `X` has to contain a "`Z bar(A)`") are easy to satisfy by just adding type-correct dummy members to the generated classes. This means that the problem in the core of JPhantom is solving the class hierarchy complementation problem, as presented in the introduction and defined rigorously in later sections. The binding of the problem to practical circumstances deserves some discussion, however.

First, note that, in our setting of the problem, we explicitly disallow modification of known code, e.g., in order to remove dependencies, or to add a supertype or a member to it. Such modifications would have a cascading effect and make it hard to argue about what properties are really preserved. Additionally, we do not assume any restrictions on the input, other than the well-formedness condition of being legal Java bytecode (according to the verifier). Strictly speaking, our well-formedness condition for the input is defined as follows: *a legal input is bytecode that can be complemented (by only adding extra class and interface definitions) so that it passes the Java verifier*. Note that this well-formedness condition does not depend on the program complement that our approach produces: an input is legal if there is *some* complement for it, not necessarily the one that JPhantom computes.

A final interesting point concerns the practical impact of the JPhantom soundness condition. For most program analyses, omitting parts of the code introduces unsoundness, if we make no other assumptions about the program or the omitted part. E.g., it is impossible to always soundly compute points-to information, or may-happen-in-parallel information when part of the program is missing. Therefore, guaranteed soundness for all clients is inherently unachievable for *any* partial program analysis approach. The practical reality is that there is a large need for facilities for handling partial programs. For instance, the Soot phantom class machinery has been one of the most common sources of discussion and questions on the Soot support lists, and it has been a central part of several Soot revisions.[4] The only "correctness condition" that Soot phantom class support is trying to achieve, however, is the low-level "the analyzer should not crash".

Given the practical interest for the solution of a worst-case unsolvable problem, we believe that our soundness guarantee makes a valuable contribution: it is much better to analyze a partial program in a way such that the Java verifier requirements (for type-level well-formedness) are satisfied than to ignore any correctness considerations, as past approaches do.

[4] Even the most recent Soot release, 2.5.0, lists improved support for phantom classes and excluding methods from an analysis as one of the major changes in the release notes.

3. Hierarchy Complementation for Multiple Inheritance

We begin with a modeling of the hierarchy complementation problem in the setting of multiple inheritance. This means that every class in our output can have multiple parents.

We can model our problem as a graph problem. Our input is a directed graph $G = (V, E)$, with two disjoint sets of nodes $V = V_{known} \dot{\bigcup} V_{phantom}$ and two disjoint sets of edges $E = E_{direct} \dot{\bigcup} E_{path}$, where $E_{direct} \subseteq V_{known} \times V$ (i.e., direct edges have to originate from known nodes—the converse is not true, as known nodes can be inferred to subtype unknown ones due to assignment instructions in the bytecode). The set of nodes V is a set of types, while the set of edges E corresponds to our subtyping constraints. That is, an edge (v_s, v_t) encodes the constraint $v_s <: v_t$. The E_{direct} subset encodes the direct-subtype constraints. The output of our algorithm should be a *DAG* (with edges from children to their parents), $G_D = (V, E')$, such that:

1. $\forall v_s \in V_{known} : (v_s, v_t) \in E' \Leftrightarrow (v_s, v_t) \in E_{direct}$ (i.e., all direct edges from known nodes are preserved and no new ones are added to such nodes)
2. $(v_s, v_t) \in E_{path} \Rightarrow$ there is a path from v_s to v_t in G_D

Note that our only limiting constraint here is that we cannot have cycles in the resulting hierarchy. Moreover, since each type may have multiple supertypes in this setting, a directed acyclic graph is fitting as our intended output.

In contrast to the general case, the problem is trivial if we have a phantom-only input, i.e., if we ignore V_{known} and E_{direct}. It suffices to employ a cycle-detection algorithm, and—if no cycles are present—return the input constraint graph as our solution: all path edges can become direct subtyping edges. If our input graph contains a cycle, then our problem is unsolvable. If not, our solution would probably contain some redundant edges (i.e., edges connecting nodes that are already connected by another path) that we could prune to minimize our output. In either case, our solution would be valid w.r.t. our constraints.

The problem becomes much more interesting when we take V_{known} into account. The source of the difficulty is the combination of cycle detection with nodes whose outgoing edge set cannot be extended. Consider first the pattern of Figure 3.

This pattern is a basic instance of interesting reasoning in the case of multiple inheritance. We have $A \in V_{known}$ such that $(A, B), (A, C), (A, D) \in E_{direct}$ and $(A, E) \in E_{path}$. We cannot, however, satisfy the path ordering constraint by adding edges to the known node A. Therefore the output

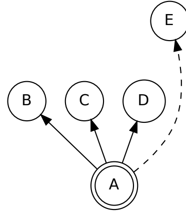


Figure 3: In any solution of these constraints, either B or C or D have to be ordered below E, since no new outgoing edges can be added to A and the path constraint to E needs to be satisfied.

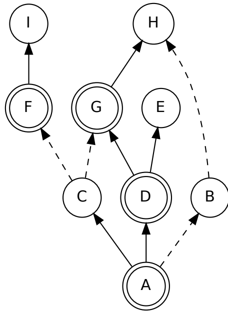


Figure 4: The phantom projection set of A is $\{C, E, H\}$. In order to satisfy path-edge (A, B) we can either add a path-edge (C, B), (E, B), or (H, B). The last one creates a cycle.

must have one of B, C, D ordered below E. We refer to the set of $\{B, C, D\}$ as the *projection set* of node A, which is a more generally useful concept.

Definition 3.1. *Projection Set.* A node $t \in V_{phantom}$ belongs to the *projection set* of a node $s \in V_{known}$ iff t is reachable from s through a path of *direct* edges.

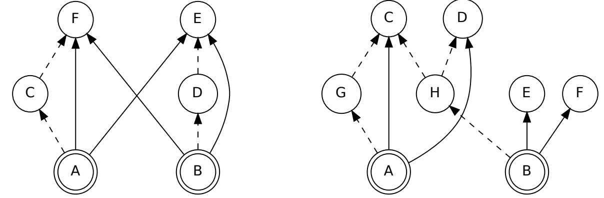
$$proj(s) \equiv \{t \in V_{phantom} : (s, t) \in {E_{direct}}^{+}\}$$

with the + symbol denoting transitive closure.

That is, for each known node we can follow its outgoing direct-edges recursively, ending each path when we reach a phantom node. For instance, in Figure 4, the phantom projection set for node A is $\{C, E, H\}$.

Referring again to Figure 4, we can see that if H is chosen from the projection set of A in order to satisfy the path-edge (A, B), and therefore edge (H, B) is added, then this would immediately create a cycle because of the existing (B, H) edge. Our algorithm should prevent such a cycle by making the correct choice from the relevant projection set.

Combining this projection set choice with cycle detection leads to interesting search outcomes. Figure 5a shows an example of unsatisfiable input. The path edge (B, D) makes either E or F be subtypes of D, and similarly the path edge (A, C) makes either E or F be subtypes of C. Nevertheless, any choice leads to cycles. In contrast, Figure 5b shows an input for which a solution is possible, and which we use to illustrate our algorithm.



(a) Unsatisfiable input.

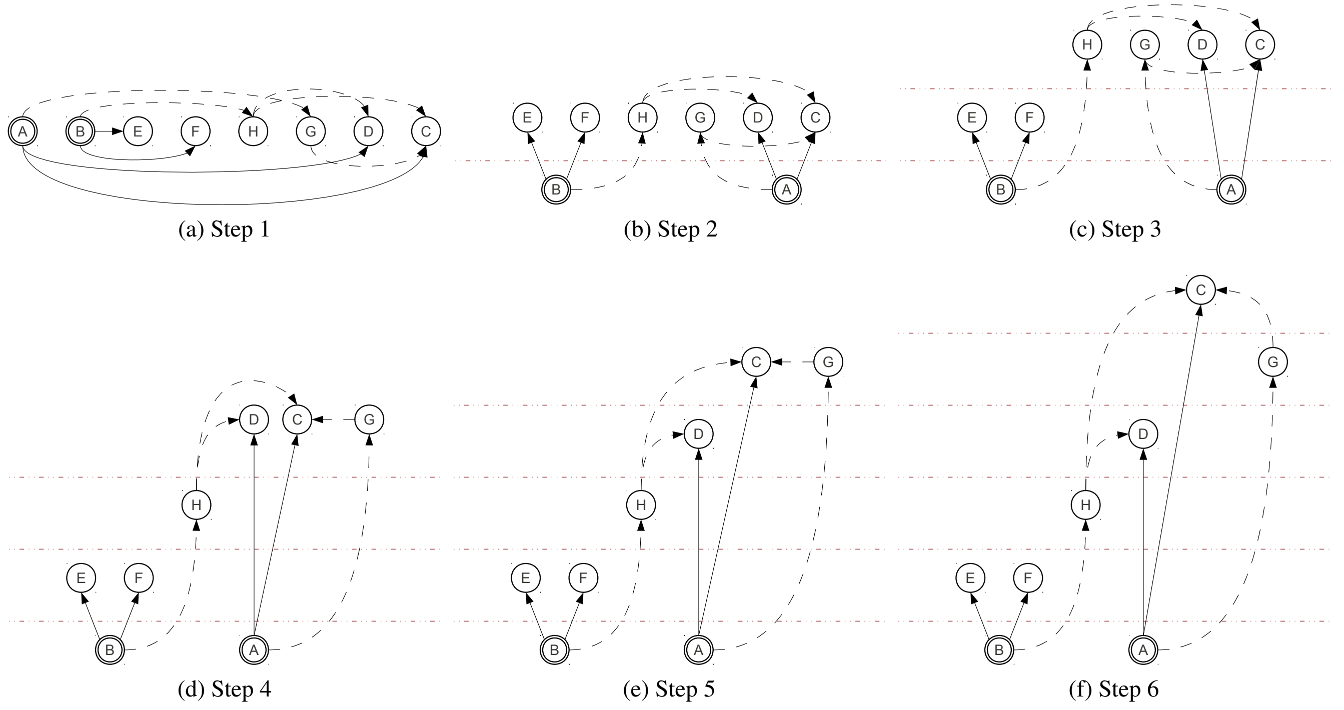
(b) Satisfiable input.

Figure 5: Multiple Inheritance Examples

Algorithm 3.1 solves in polynomial time (an easy bound is $O(|V| \cdot |E|)$) any instance of the hierarchy complementation problem in the multiple inheritance setting. The main part of the algorithm is function STRATIFY(), which computes a stratification with the property that any constraint edge is facing upwards (i.e., from a lower to a higher stratum). Moreover, this stratification ensures that, for any path-edge (s, t) originating from a known node, there will exist a phantom node p in the projection set of s that is placed lower than t. Given this stratification, it is easy to compute the final solution (as in function SOLVE()). To satisfy any such path-edge (s, t), we add a direct-edge from p to t. This respects our invariant of all edges facing upwards, thus ensuring that no cycles will be present in our solution.

Function STRATIFY() starts from a single stratum, and then computes on each iteration a new stratification, S_{i+1}, by building on the stratification of the previous step, S_i, and advancing some nodes to a higher stratum in order to satisfy constraints. This process is repeated until we converge to the final stratification, which will respect all of our constraints (line 21). If no new node converges at some step (i.e., all nodes that reached a certain stratum advance to the next), then we can be certain that we are dealing with unsatisfiable input, and terminate, thus avoiding infinite recursion (line 23). The nodes to be advanced at each step are determined at line 18, which captures the essence of the algorithm. The new stratum of a node t will be either (i) its current stratum, (ii) the stratum right above the source of an edge (s, t), or (iii) the one right above the *lowest* projection node of the source of a path-edge (s, t) originating from a known node—whichever is higher. These conditions raise the stratum of a node to the minimum required to satisfy the natural constraints of the problem, per our above discussion: edges in the solution should be from lower to higher strata.

Figure 6 presents an illustration of the algorithm's application to the example of Figure 5b. The sets $\{C, D\}$ and



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

Figure 6: An example of the stratification produced by the multiple-inheritance solver for Example 5b.

Algorithm 3.1 Multiple-inheritance solver

1: **function** SOLVE($G = (V, E)$)
2: $\quad S \leftarrow$ STRATIFY(G)
3: $\quad U \leftarrow \{(s, t) \in E_{path} : s \in V_{known}\}$
4: $\quad E_S \leftarrow E \setminus U$
5: $\quad$ **for all** $(s, t) \in U$ **do**
6: $\quad\quad$ let $p \in proj(s) : S[p] < S[t]$ ▷ such p always exists
7: $\quad\quad E_S \leftarrow E_S \cup \{(p, t)\}$
8: $\quad$ **end for**
9: $\quad$ **return** E_S
10: **end function**
11: **function** STRATIFY($G = (V, E)$)
12: $\quad U \leftarrow \{(s, t) \in E_{path} : s \in V_{known}\}$
13: $\quad$ **for all** $t \in V$ **do**
14: $\quad\quad S_0[t] \leftarrow 0$
15: $\quad$ **end for**
16: $\quad$ **for** $i = 0 \rightarrow |V| - 1$ **do**
17: $\quad\quad$ **for all** $t \in V$ **do**
18: $$S_{i+1}[t] \leftarrow \max \left\{ \begin{array}{c} S_i[t] \\ \max\limits_{(s,t)\in E} \{1 + S_i[s]\} \\ \max\limits_{(s,t)\in U} \{1 + \min\limits_{p\in proj(s)} \{S_i[p]\}\} \end{array} \right\}$$
19: $\quad\quad$ **end for**
20: $\quad\quad$ **if** $\forall v \in V : S_{i+1}[v] = S_i[v]$ **then**
21: $\quad\quad\quad$ **return** S_i ▷ reached a fixpoint
22: $\quad\quad$ **else if** $\forall v \in V : S_{i+1}[v] = S_i[v] \Rightarrow S_i[v] = S_{i-1}[v]$ **then**
23: $\quad\quad\quad$ **break** ▷ no progress made on this step
24: $\quad\quad$ **end if**
25: $\quad$ **end for**
26: $\quad$ **return** error ▷ unsolvable constraint graph
27: **end function**

$\{E, F\}$ are the projection sets of nodes A and B respectively. At the first step, all nodes will be placed in the lowest stratum. Note that, at this point, all nodes could be placed in topological order: Figure 6a is perfectly valid as the output of a topological sort. However, this is not a solution by our standards, since node A cannot satisfy the edge to G because both of its projection nodes, D and C, are placed after G. Adding an edge from either one would be subject to creating cycles. At the next step, our algorithm advances every node except A and B, since all are edge targets. At step 3, things become more interesting. Nodes D, C have to be advanced by the same criterion, since node H contains edges to both, and they all reside in the same stratum at step 2. However, nodes H and G have to be advanced for a different reason, since they are targets of path-edges originating from known nodes, namely A and B, whose projections ($\{D, C\}$ and $\{E, F\}$ respectively) were on the second stratum during the previous step. At step 4, this condition ceases to exist for node H, since nodes E, F have "stabilized" at a lower stratum. This in turn causes node D to stabilize at step 5. At step 6, G can also stay put, since it is in a higher stratum than the lowest projection of A, namely D. No nodes are advanced at step 7 (which is omitted in Figure 6), thus signifying that our stratification has successfully converged to its final form. It is therefore simple to compute a solution, by adding edges $(H, D), (H, C), (G, C)$, (D, G) and either (F, H) or (E, H) to the direct-edges $(A, C), (A, D), (B, E), (B, F)$. This set of edges will constitute our final solution.

It is also easy to see that our algorithm would soundly detect that the example of Figure 5a is unsatisfiable. At

the first step, only known nodes A, B would remain in the lowest stratum, but on the next iteration all remaining nodes would advance again, thus triggering the condition of failure (line 23), since an iteration passed with no progress made.

A detailed proof of the correctness of our algorithm can be found in Appendix A.

4. Hierarchy Complementation for Single Inheritance

The problem for a single inheritance setting has a very similar statement as in the earlier case of multiple inheritance, but markedly different reasoning intricacies and solution approaches, due to a newly arising constraint: every class in this setting can only have a single parent.

Formally, our problem is modeled in much the same way as before. Our input is again a directed graph $G = (V, E)$, with two disjoint sets of nodes $V = V_{known} \dot{\bigcup} V_{phantom}$ and two disjoint sets of edges $E = E_{direct} \dot{\bigcup} E_{path}$, where $E_{direct} \subseteq V_{known} \times V$. The difference is that the output of our algorithm should be a directed *tree* (instead of a DAG), $G_T = (V, E')$, such that the same conditions as in the earlier case are satisfied:

1. $\forall v_s \in V_{known} : (v_s, v_t) \in E' \Leftrightarrow (v_s, v_t) \in E_{direct}$
2. $(v_s, v_t) \in E_{path} \Rightarrow$ there is a path from v_s to v_t in G_T

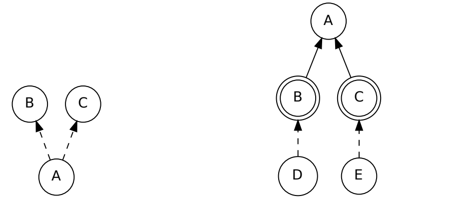
Without loss of generality, we assume that there exists a "root" node $n_r \in V_{known}$ that is a common supertype for all of our types. If no such type exists, we can create an artificial one, by adding extra constraint edges. In this way, we can be certain that computing a graph with a single outgoing edge for all nodes (but one) will form a tree instead of a forest.

The problem is quite hard in its general setting. There are several patterns that necessitate a complex search in the space of possibilities. Figures 7a-7d show some basic patterns that induce complex constraints. All nodes reachable from a single one need to be linearly ordered (Figure 7a shows the simplest case). This requires computing an ordering (i.e., guessing a permutation) of these nodes. Other constraints can render some of the permutations invalid. The basic pattern behind such restrictions is that of Figure 7b: there are hierarchies that cannot be related. Combining the two patterns suggests that there needs to be a search in the space of permutations for a valid one: Figures 7c and 7d show some simple cases.

Composing such constraints into more complex hierarchies gives an idea of the difficulty of the search involved. Figure 8 shows an example where it is hard to see without complex reasoning which of the E, F, G nodes have to be placed above A and which cannot.

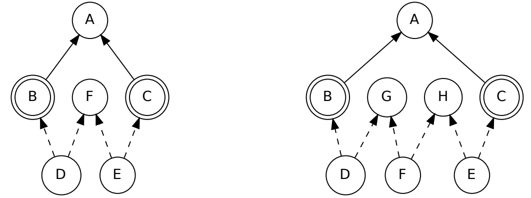
Clearly the problem can be modeled as a constraint satisfaction problem instance, where $V_{phantom}$ is our set of variables and V is the domain of values (representing the variable's direct supertype). The path-edges and the absence of cycles constitute our constraints. This requires an exponential search in the worst case. Indeed, our implementation performs precisely such an exhaustive search, but with a heuristic choice of nodes so that the search tries to satisfy the constraints introduced by the patterns in Figures 7a and 7b—i.e., the pattern of Figure 7a is identified, all induced permutations are tried, and the pattern of Figure 7b is used to prune them eagerly, instead of waiting to detect failure later.

Most importantly, our approach provides special handling for a simple but practically quite common case. In this special case, there is a polynomial algorithm for solving the problem and exhaustive search is avoided.



(a) B and C must be subtype-related (in either direction).

(b) D and E cannot be subtype-related.



(c) A has to be a subtype of F.

(d) A has to be a subtype of either G or H.

Figure 7: Single Inheritance Basic Patterns

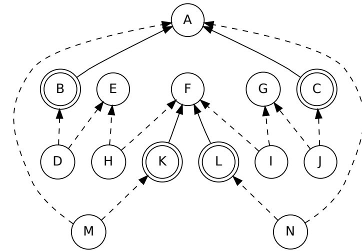
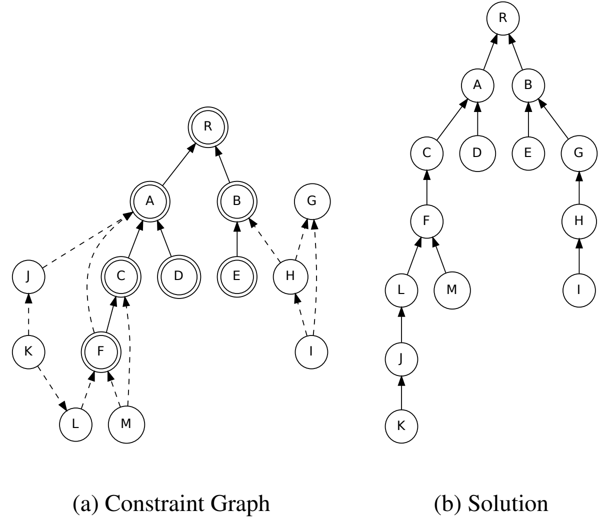


Figure 8: Harder composite example of single-inheritance constraints. The (undirected) path from B to C through E, F, G implies that $(A <: E) \vee (A <: F) \vee (A <: G)$. However, since F is the first common known supertype of M and N, and A just a supertype of both, $F <: A$, and thus $(A <: E) \vee (A <: G)$.

Algorithm 4.1 Single-inheritance solver for strictly known direct-supertypes

1: **function** SOLVE($G = (V, E)$)
2: let R be the "*root*" node of V
3: let S be the tree of known nodes (V_{known}, E_{direct})
4: **for all** $(s, t) \in E_{path} : s \in V_{known}$ **do**
5: **if** $\nexists$ path $s \rightsquigarrow t$ in S **then**
6: **return** error (unsatisfiable constraint)
7: **end if**
8: $E_{path} \leftarrow E_{path} \setminus \{(s, t)\}$ ▷ remove already satisfied edge
9: **end for**
10: **for all** $v \in V_{phantom}$ **do**
11: MAKESET(v) ▷ create single-element disjoint sets
12: **end for**
13: **for all** $(s, t) \in E_{path} : t \in V_{phantom}$ **do**
14: UNION(s, t) ▷ merge two connected (phantom) components
15: **end for** ▷ result: undirected connected components (UCCs)
16: **for all** $v \in V_{phantom}$ **do**
17: $k \leftarrow$ FIND(v)
18: $top[k] \leftarrow R$ ▷ initially "root"
19: **end for** ▷ init UCC's lowest common known superclass (LCS)
20: **for all** $(s, t) \in E_{path} : t \in V_{known}$ **do** ▷ must be $s \in V_{phantom}$
21: $k \leftarrow$ FIND(s)
22: **if** $\exists$ path $t \rightsquigarrow top[k]$ in S **then**
23: $top[k] \leftarrow t$ ▷ lower superclass found, update LCS
24: **else if** $\nexists$ path $top[k] \rightsquigarrow t$ in S **then**
25: **return** error (unsatisfiable constraint)
26: **end if**
27: **end for**
28: **for all** $k \mapsto v$ in *top* **do** ▷ for each UCC and its LCS
29: $U \leftarrow \{(s, t) \in E_{path} : t \in V_{phantom} \land \text{FIND}(s) = k\}$
▷ directed subgraph of original over nodes of this UCC
30: $L \leftarrow$ a topological order of U ▷ linearize subgraph
31: $hd \leftarrow$ the top node of L
32: $S \leftarrow S \cup L \cup \{(hd, v)\}$
33: **end for**
34: **return** S
35: **end function**

Simplified setting: No direct-edges to phantom nodes. It is easy to solve the problem in the case that there are no direct edges from known nodes to phantom nodes. Since we are in a single-inheritance setting, this means that no class in the known part of the program has a superclass in the complement that we are trying to produce. In this case, we have that $E_{direct} \subseteq V_{known} \times V_{known}$. The extra condition allows us to employ a fast polynomial time algorithm. This interesting case of our problem is very common in practice. Intuitively, the ease of dealing with this case stems from avoiding the search in the space of permutations when the input contains patterns such as those in Figure 7c: if two permutations have elements in common (e.g., the permutation of B and F, and that of F and C in Figure 7c) they cannot include nodes that are guaranteed to be subtype-unrelated (such as B and C in this example) and all unknown nodes have to be below the known ones in any solution.



(a) Constraint Graph (b) Solution

Figure 9: Algorithm 4.1 - Example

Algorithm 4.1 first removes path-edges originating from known-nodes, after verifying that the corresponding paths indeed exist. It then uses union/find data structures to compute connected components of phantom nodes, while treating path-edges as *undirected* edges: anything connected through such edges can safely end up in a single linear ordering. Then, for each phantom undirected connected component, it computes the lowest known-node to serve as the first-common-supertype of all of this component's phantom nodes. Note that when two known-nodes are reachable by two phantom nodes of the same connected component (in the phantom subgraph), then one of them ought to be a supertype of the other, or else no solution can exist in a single inheritance setting. This condition is captured in line 24. After the first common (known) supertype for every connected component has been computed, a mere topological sort, i.e. placing all relevant nodes in a total order, is enough to satisfy all of this component's constraints. This may introduce many superfluous edges in the solution: these edges are not actually required by our constraints (since a topological order is a total order). In practice, we produce a partial order by using a variant of topological sort that generates a tree instead of a list as its result, but a full topological sort also satisfies the correctness requirements of the algorithm. (We return to the topic of why we actually want a weaker ordering in Section 5.)

In the example of Figure 9, Algorithm 4.1 first checks and removes the (F, A) path-edge. Then the phantom nodes are divided in the following phantom connected components: $\{G, H, I\}$, $\{J, K, L\}$, and $\{M\}$. The first common known supertype for each component is B, F, and F respectively. Each component is then linearized, which generates the following complete orders that are appended to the output: $I <: H <: G <: B$, $K <: J <: L <: F$, and $M <: F$.

5. Single Inheritance, Multiple Subtyping: Classes and Interfaces

It is easy to combine the single- and multiple-inheritance approaches of the last two sections in the context of a language that has single inheritance but multiple subtyping. It is a common case for strongly-typed languages to allow multiple inheritance only for a subset of types. Java and C# interfaces [10, 12], and Scala traits [17] are such examples.

In order to support such a separation, we have to introduce a new dimension to our problem that can be simulated as a graph coloring variant. Each node in V can be assigned a color denoting its inheritance type. A *black* node can have many direct supertypes (i.e., multiple inheritance), while a *white* node can only have one (i.e., single inheritance). We will use the terms “white node” (resp. “black node”) and “class” (resp. “interface”) interchangeably.

Note that, initially, our input may not fully determine the final color for each of its types. Thus, we have to introduce a new color (*grey*) to refer to the subset of nodes whose color is yet undetermined. In the end, our solution should soundly determine a safe color (black or white) for each of the (grey) input nodes, so that no constraints of the verifier will be violated.

Therefore, our solution in this new setting is a synthesis of a single inheritance and a multiple inheritance solution. That is, the output of our algorithm should be a *DAG* that satisfies the same conditions as those in the multiple inheritance setting, $G_S = (V, E')$, and a function $f_c : V \rightarrow \{black, white\}$, such that the restriction of G_S to $\{v \in V : f_c(v) = white\}$ (i.e., white nodes) is a tree.

To safely decompose our problem into two different subproblems (one for single and one for multiple inheritance), we assign colors to all nodes as a preprocessing step. There are two kinds of constraints that lead to restricting the colors of a node. First, we have local constraints: we may get a node color from the initial input—i.e., an observed bytecode instruction (such as `invokeinterface`) may directly restrict the color of a phantom type. (More constraints of this form are discussed in Section 6.) Second, we may get transitive constraints, due to restrictions on subtyping. Interfaces can only subtype interfaces (except for the `Object` class in Java). This leads to two types of transitive constraints: If a black node s has a path to node t, then t must also be black (interfaces can only extend interfaces). Symmetrically, if a node s has a path to a white node t, then s must also be white (classes can only be extended by other classes).

Furthermore, phantom nodes with no color constraints can be safely assumed to be interfaces (black), for maximum flexibility in solving other constraints. It is always easier to satisfy a given set of constraints in a multiple inheritance setting instead of a single inheritance setting, since the conditions of single inheritance are stricter (a tree is a DAG).

As a result of the above observations, we can color all nodes by applying local or transitive constraints to the original input before solving a single and a multiple inheritance hierarchy complementation problem separately. That is, we can follow every possible path from any node whose color has already been set and mark the nodes we find along the way accordingly. The color of our source node determines the direction of movement (i.e., from white source nodes, we have to go backwards). When this process is over, we can assign the color black to all remaining undetermined (in terms of color) nodes. An example of this process can be seen in our earlier Figure 2. Once we have assigned a *black-or-white* color to every node, we can split our constraint graph into two subgraphs by isolating white-to-white edges (and feeding them to a single inheritance solver). After we have determined our class hierarchy, we can proceed with satisfying the rest of the edges using multiple inheritance rules.

The key to this approach is that the single inheritance solver does not need the output of the multiple inheritance solver to compute a solution, and vice versa. All we need to ensure (for the multiple inheritance solver) is that we take into account class supertypes that are reachable through direct edges of a known class when determining the class’s *projection set*. Thus, the class/interface decomposition indeed produces two independent subproblems that can be solved separately. The composition of the two solutions will certainly not create any cycles, if its two subparts do not contain any. If that was not the case, then there would be a cycle that contained at least one class and one interface, which is impossible since no interface can be a subtype of a class (other than `Object`) in Java.

As for our arbitrary choice of defaulting undetermined nodes to interfaces, suppose that a solution exists if a subset U of those undetermined nodes were treated as classes. We could then transform this solution to another one where these nodes were interfaces instead. The single inheritance solution could be produced by replacing each node in U with its parent (in the former single inheritance solution), w.r.t. its incoming edges, and then removing it, until no nodes in U were present. This process would still satisfy all constraints on the remaining class nodes. A multiple inheritance solution also exists. Consider the union of the former multiple plus single inheritance solution. The result is a DAG that respects all of the multiple inheritance setting constraints. Again, we can erase any edges to class-determined nodes (i.e., all class nodes that are not in U) in a way that all subtype relations involving the rest of the nodes remain unaltered, i.e., by iteratively replacing an edge to a class-determined node with edges to all of its direct supertypes, until no edges to class-determined nodes are left. This process would yield a valid multiple inheritance solution that can be safely combined with the single inheritance one. Therefore, marking undetermined nodes as interfaces does not affect the outcome of our algorithm, i.e., no solution will be found if and only if no solution existed.

6. Implementation and Practical Evaluation

We next discuss practical aspects of our implementation. First, we consider the *program analysis* part of our work, which solves the problem of producing complements of a partial Java program by appealing to the solver of the class hierarchy complementation problem. Subsequently, we present experiments applying our JPhantom tool to real programs.

6.1 JPhantom Implementation

JPhantom is a practical and scalable tool for program complementation, based on the algorithms we have presented in this paper. [5] JPhantom uses the ASM library [7] to read and trasform Java bytecode. Given a jar file that contains phantom references, it produces a new jar file with dummy implementations for each phantom class. The resulting jar file satisfies all formal constraints of the JVM Specification [14]. We give a brief explanation of the different stages of computation for the analysis of an input jar file by JPhantom.

JPhantom execution consists of the following steps. It (1) performs a first pass over the jar contents in order to recreate the existing class hierarchy (type signatures only) and store the field and method declarations of the contained classes, then (2) makes a second pass to extract all phantom references and store the full class representations. A third pass (3) extracts all relevant type constraints, before (4) they are fed to JPhantom's hierarchy complementation solver, which computes a valid solution, if such a solution is possible. At this point, we can proceed to (5) bytecode generation, where we create new class files for our missing (phantom) types. Finally, we compute method bodies to add to each type. For instance, when the solver determines that a phantom-class type X must implement an interface type Y, all missing methods of Y should be added to X, so that the resulting bytecode is valid. After all such methods have been computed, they are added in the last (6) step of execution.

Phantom references include references to missing classes, as well as references to missing fields and methods. Note that both phantom and existing classes may have references to missing members, since there are cases of existing classes calling a method or referencing a field declared in one of their phantom supertypes. JPhantom detects all such references and adds the relevant missing declarations to its output. If a member is missing from a phantom class, we add it directly to that class as part of JPhantom's output. Otherwise, if a member is missing from an existing class, we add it to an appropriate phantom supertype in its projection set instead. We encode these declarations as additional constraints over the missing classes, generated in the second step of JPhantom's execution. It suffices to use the existing class hierarchy and declared members (step 1), to perform member lookup for the purpose of determining if a member is missing and where it should be added.

[5] JPhantom is available online at `https://github.com/gbalats/jphantom`.

The most interesting aspects of the above steps have to do with analyzing the bytecode to produce the constraints (step 3) used as input to the hierarchy complementation algorithm. In order to extract type constraints, we have to simulate a symbolic execution of Java bytecode by following every possible execution path, while computing the types of stack and local variables. This is necessary because, in general, bytecodes receive some untyped arguments whose types we need to infer, in order to extract our constraints. This process is analogous to Pass 3 [14, Section 4.9.2] of the bytecode verification process.

When computing such type information for stack and local variables, there are points where we have to merge two different paths of execution. That is, the two paths may map the same variable to different types, in which case we have to merge two different types into a new one. Typically, when merging two types A, B the resulting type is the first common superclass of A and B. In Java, there always exists such a common superclass since every reference type (interfaces included) is a subtype of `java.lang.Object`.

In our case, however, since we do not have the complete type hierarchy at the time of constraint extraction, we cannot compute the first common superclass for any two nodes. This is why we apply the alternative technique of storing *sets of reference types*, as presented in alternative verifier designs [18]. I.e., our bytecode analyzer stores not a single type, but a set of types for each variable at every point of execution. Figure 10 lists the constraints that may be generated by the analyzer for certain bytecodes. Since our analyzer generates constraints due to widening reference conversions, it is easy to see that storing a set of reference types fits our needs well. Consider the following case:

```
class Test {
   A foo(B b, C c) {
      return (b == null) ?
         c : b;
   }
}
```

```
A foo(B, C);
  Code:
   0:       aload_1
   1:       ifnonnull 8
   4:       aload_2
   5:       goto 9
   8:       aload_1
   9:       areturn
```

Our analyzer will compute that the stack contains a single item with type $\{B, C\}$, before position 9, which is the outcome of merging the two different execution paths. Let us also assume that A and B are phantom classes. This toy example demonstrates why we have chosen to store sets of types, since we cannot compute the first common superclass of B, C. After our tool has completed the analysis of method `foo()`, it will generate (because of the ARETURN instruction) the constraint $B <: A \wedge C <: A$.

Opcode	*Types*	*Stack Types*	*Constraints*
AASTORE		$a : E[]\ \ i : int\ \ v : V$	$V <: E$
ARETURN		$obj : S$	$S <: R_m$
ASTORE	T	$obj : S$	$S <: T$
ATHROW		$obj : S$	$S <:$ "java.lang.Throwable"
GETFIELD	$T.F$	$obj : S$	$isClass(T) \wedge S <: T$
PUTFIELD	$T.F$	$obj : S\ \ v : U$	$isClass(T) \wedge S <: T \wedge U <: F$
PUTSTATIC	$T.F$	$v : U$	$isClass(T) \wedge U <: F$
INVOKEINTERFACE	$T.(\overline{A})R$	$arg_0 : S_0\ \ arg_1 : S_1 \ldots$	$isIface(T) \wedge S_0 <: T$
INVOKEVIRTUAL	$T.(\overline{A})R$	$arg_0 : S_0\ \ arg_1 : S_1 \ldots$	$isClass(T) \wedge S_0 <: T$
INVOKESPECIAL	$T.(\overline{A})R$	$arg_0 : S_0\ \ arg_1 : S_1 \ldots$	$name =$ "<init>" $\Rightarrow isClass(T) \wedge S_0 <: T$
INVOKESTATIC	$T.(\overline{A})R$	$arg_1 : S_1 \ldots$	$isClass(T)$
INVOKE*	$T.(\overline{A})R$	$(arg_0 : S_0)\ \ arg_1 : S_1 \ldots$	$S_i <: A_i, \forall i = 1, \ldots$

Figure 10: *Generated Bytecode Constraints*. At this point, our analyzer has already computed the (sets of) types for every stack and local variable at every point of execution (bytecode in method). For simplicity, we assume that each set of reference types contains a single element (3rd column). Each bytecode may involve some declared types (2nd column) by references in the constant pool or by entries in the local variable table (if such exists). Also, let R_m be the containing method's return type.

6.2 JPhantom in Practice

We next detail a typical usage scenario of JPhantom, together with the complications that would arise in its absence.

Consider performing a static analysis of a large Java program. For instance, the Doop framework [6, 13] integrates points-to analysis with call-graph construction, computation of heap object points-to information, and various client analyses (escape analysis, virtual call elimination, class cast elimination). Doop uses Soot as a front-end and post-processes the facts generated by Soot. When faced with an incomplete program, the user of the analysis is faced with various issues. To illustrate and quantify them we created a synthetic incomplete program, *antlr-minus*, by artificially subtracting parts of the antlr parser generator jar. (We also use *antlr-minus* as a performance benchmark in the next section.)

A user that tries to analyze *antlr-minus* will encounter the following issues:

- *Crash in Soot.* Earlier versions of Soot, e.g., Soot 2.3.0, will often crash when trying to analyze a program that contains phantom references. Soot provides the `-allow-phantom` flag, as a command-line option that the user can set to inform Soot that its input contains phantom references, and that Soot should try to handle them instead of terminating with an error. However, for several Soot versions the flag is not sufficient to prevent Soot from crashing in some cases.
- *Need to handle phantom references in the client of Soot.* Although the latest version of Soot (2.5) has increased its tolerance of phantom references to the point where it no longer crashes, this only prevents against crashes in Soot itself and does not yield any meaningful handling of phantom references. The problem is propagated to the client of Soot. The client analysis (any external tool that uses Soot) now needs to have special-case code for handling phantom classes, in whichever way makes sense to the client. There is no evident general-purpose solution to fixing the Soot output for *any* client without adding code to deal with phantom references, essentially duplicating what JPhantom does already. In our case, if the Doop front-end that reads Soot information tried to just handle phantom references as regular references, it would crash (as we have confirmed experimentally), since it needs to encode for every variable its full type information (e.g., member methods). (The Doop front-end does not crash in practice because it handles phantom references specially, by merely ignoring them, as we discuss next.) In contrast, JPhantom allows any tool completely unaware of phantom references, such as the Doop front-end, to be able to run without unexpected behavior, as long as its input is first transformed by JPhantom.
- *Incompleteness when analyzing with Doop.* The Doop front-end is coded so that it avoids crashes but only at the cost of completely ignoring any reference to a phantom class. A method that takes phantom types as arguments is just skipped. This handling has been the default for Doop since its original version. Unfortunately, this leads to incompleteness in the resulting analysis performed by Doop.

Figure 11 presents a Venn diagram over the sets of reachable methods as computed by Doop for three different inputs: (i) the original *antlr* jar, (ii) our synthetic benchmark, *antlr-minus*, and (iii) the output of JPhantom after analyzing *antlr-minus*, that is, a transformed version of

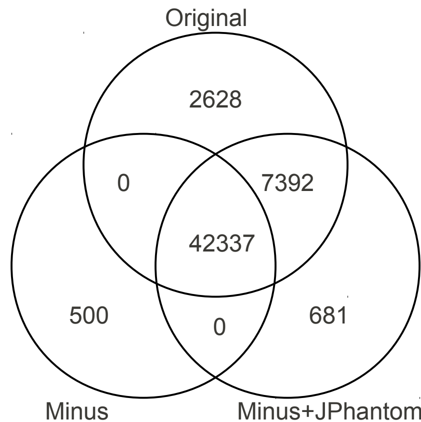


Figure 11: A Venn diagram that shows how three different sets of reachable methods relate to each other. These three sets—(i) *Original*, (ii) *Minus*, and (iii) *Minus+JPhantom*—correspond to the outcomes of analyzing (i) the *antlr* jar (original), (ii) the *antlr-minus* jar (subset of the original jar), and (iii) the *antlr-minus* jar after being transformed by JPhantom, respectively. The sets are not drawn to scale: the size of each subset is indicated only by the number in it.

the *antlr-minus* jar with no phantom references. The original jar yields $52,357$ reachable methods, out of which only $42,337$ are detected in the presence of phantom references (*antlr-minus*), without using JPhantom. Additionally, phantom references introduce 500 false positives that correspond to non-existing methods.[6] After employing JPhantom to alleviate the effect of phantom references, Doop manages to find $7,392$ of the $10,020$ missing reachable methods, resulting in 73.77% recall (over the missing methods alone, or 95% over all methods). The false positives of directly analyzing *antlr-minus* disappear but 681 new ones emerge, yielding a precision of 98.65%. Even so, this allows us to discover almost 3 out of every 4 missing reachable methods, which originally constituted 19.14% of the total reachable methods, dropping this percentage to just 5.02%.

It is notable that this high recall is achieved although recall could, in principle, be arbitrarily low. JPhantom is trying to guess the structure of missing code with as much information as remains in existing code—but this could be a tiny fraction of the missing information. The missing code could be hiding a huge portion of the application, and expose only a handful of phantom types on the unknown/known code boundary.

In summary, JPhantom avoids problems with crashes when encountering phantom references as well as incompleteness when phantom references are merely ignored. In practice, it is effective in discovering large parts of the interface for missing methods and the produced complement respects the requirements of the Java VM verifier, i.e., the most fundamental Java well-formedness rules for types.

6.3 Performance Experiments

We use a 64-bit machine with a quad-core Intel i7 2.8GHz CPU. The machine has 8GB of RAM. We ran our experiments using JDK 1.7 (64-bit).

Our benchmarks consist of (1) *antlr*, a parser generator, (2) *antlrworks*, the GUI Development Environment for antlr, (3) *c3p0*, a library that provides extensions to traditional JDBC drivers, (4) *jruby* and (5) *jython*, implementations of Ruby and Python programming languages respectively that run on top of the JVM, (6) logback-classic and (7) logback-core, two modules of the *logback* logging framework, (8) *pmd*, a Java source code analyzer, (9) *postgres*, the PosgreSQL JDBC driver, (10) *sablecc*, a compiler generator, and (11) *antlr-minus*, a synthetic benchmark described in the previous section. Every benchmark is just a jar file that serves as JPhantom's input, which then detects all phantom references and generates the complemented jar.

We encountered most of these benchmarks in our own work doing static program analysis with the Doop framework [6, 13]. For many of the benchmarks it was, upon original encounter, an unexpected discovery that they could not be analyzed due to dependencies to unknown classes in other libraries.

Figure 12 presents input features and the running time of JPhantom for each of our benchmarks. The first column is the name of the benchmark. The second column is the size of the output, i.e., the complemented jar, divided into the original size of the input (benchmark) and the size of the complement itself (i.e., the size of the generated phantom classes). The third and fourth columns are the number of phantom classes and constraints detected respectively. The last column is the running time of JPhantom, including the time to analyze the input, compute a type hierarchy that respects all of the constraints detected, create the phantom classes with the required members and supertypes, augment the input jar and flush its contents to disk.

Even the largest benchmark (jruby) takes seconds to complete. Moreover, the size of the input is highly correlated with the running time of JPhantom and much less correlated with the number of constraints. This suggests that most of the time is spent on reading and analyzing the input, rather than on the type hierarchy solver. The only slight exception is the logback-classic benchmark, which requires about 1.8 seconds to complete despite its small size. This is due to the large number of phantom classes and constraints this benchmark produces, which is to be expected since it is built on top of logback-core (which is not supplied as part of the input). This practice of creating such a strong dependency is probably justified by logback's design. The framework im-

[6] It may seem surprising that eliminating code can introduce new (falsely) reachable methods. The reason is that a non-existent method m in class C may be reported reachable, based on method signature information on the call-site alone, whereas in the original code the true reachable method m was defined in a, now missing, superclass S of C, and not in C.

Input jar	*Size*	*Phantom*	*Constraints*	*Time*
antlr	3.3M + 0.7K	1	2	4.82*s*
antlrworks	3.5M + 2.2K	5	7	6.11*s*
c3p0	597K + 1.8K	4	2	2.05*s*
jruby	19M + 5.9K	16	20	13.70*s*
jython	2.5M + 4.0K	8	9	3.26*s*
logback-classic	247K + 55K	148	212	1.76*s*
logback-core	358K + 7.9K	22	22	1.61*s*
pmd	1.2M + 11K	28	36	2.62*s*
postgres	499K + 0	0	0	1.95*s*
sablecc	306K + 2.3K	5	8	1.59*s*
antlr-minus	3.2M + 17K	37	103	5.82*s*

Figure 12: Results of experiments.

plements the SLF4J (Simple Logging Facade for Java) protocol, which acts as a common interface for a variety of logging frameworks, and hides the actual framework (called *binding*) to be used underneath. From both logback-classic and antlr-minus we can see that JPhantom scales well as the number of constraints increases.

To see the constraints and their solution for a benchmark instance, consider the list below, which is the actual execution output of a JPhantom run on the jruby benchmark:

```
Phantom Classes Detected:                           [constraint]

org.apache.tools.ant.BuildException          must be a class
org.apache.tools.ant.Task                    must be a class
org.apache.tools.ant.Project
org.apache.bsf.util.BSFFunctions             must be a class
org.apache.bsf.util.BSFEngineImpl            must be a class
org.apache.bsf.BSFException                  must be a class
org.apache.bsf.BSFManager                    must be a class
org.apache.bsf.BSFDeclaredBean               must be a class
org.apache.bsf.BSFEngine
org.osgi.framework.Bundle                 must be an interface
org.osgi.framework.BundleReference
org.osgi.framework.FrameworkUtil             must be a class
org.osgi.framework.BundleException
org.osgi.framework.BundleContext          must be an interface
java.dyn.Coroutine                           must be a class
java.dyn.CoroutineBase

Constraints:

org.apache.bsf.BSFException <: Throwable
org.apache.tools.ant.BuildException <: Throwable
org.osgi.framework.BundleException <: Throwable
org.jruby.embed.bsf.JRubyEngine <:
  org.apache.bsf.util.BSFEngineImpl
org.jruby.embed.bsf.JRubyEngine <:
  org.apache.bsf.BSFEngine
org.jruby.ant.RakeTaskBase <: org.apache.tools.ant.Task
org.jruby.javasupport.bsf.JRubyEngine <:
  org.apache.bsf.BSFEngine
org.jruby.ext.fiber.CoroutineFiber$1 <:
  java.dyn.Coroutine
org.jruby.javasupport.bsf.JRubyEngine <:
  org.apache.bsf.util.BSFEngineImpl

Class Hierarchy
* class java.lang.Object
  * class org.apache.bsf.BSFManager
  * class org.osgi.framework.FrameworkUtil
  * class Throwable (implements java.io.Serializable)
    * class org.osgi.framework.BundleException
    * class org.apache.tools.ant.BuildException
    * class org.apache.bsf.BSFException
  * class org.apache.bsf.BSFDeclaredBean
  * class org.apache.bsf.util.BSFFunctions
  * class org.apache.tools.ant.Task
    * class org.jruby.ant.RakeTaskBase
  * class java.dyn.Coroutine
    * class org.jruby.ext.fiber.CoroutineFiber$1
  * class org.apache.bsf.util.BSFEngineImpl (implements
     org.apache.bsf.BSFEngine)
    * class org.jruby.javasupport.bsf.JRubyEngine
    * class org.jruby.embed.bsf.JRubyEngine

Interface Hierarchy
* interface org.osgi.framework.Bundle
* interface org.apache.bsf.BSFEngine
* interface java.io.Serializable
* interface org.osgi.framework.BundleContext
```

It is evident that the final hierarchy respects all of the reported constraints. Some interesting points are that: (i) `org.apache.bsf.BSFEngine` defaults to interface since no constraint determines whether it is actually an interface or a class, (ii) `org.osgi.framework.BundleException` is inferred to be a class since it is a subtype of the class `Throwable`, and (iii) two known classes, `org.jruby.javasupport.bsf.JRubyEngine` and `org.jruby.embed.bsf.JRubyEngine`, used as subtypes of interface `org.apache.bsf.BSFEngine`, add the latter to the supertypes of their phantom projection, `org.apache.bsf.util.BSFEngineImpl`.

7. Discussion

We next discuss the problem of hierarchy complementation speculatively, in settings different from ours. The general problem is that of complementing programs so that they respect static well-formedness requirements. Thus, the problem applies to language-level type systems, static analyses (e.g., "complement this program so that it passes this analysis, defined a priori") and other settings more general than our Java bytecode domain. Indeed, much of our ability to solve the problem effectively has to do with the simple type checking performed by the Java bytecode verifier. The verifier effectively checks monomorphic types, i.e., that a reference to an object is statically guaranteed to refer to memory with the expected layout.

If we were to transpose the problem to the domain of Java source code, the constraints to be derived are richer and more complex than the ones we encountered. The Java language-level type system has intricate requirements relative to over-

riding, casts, exceptions, and more. By way of example, we discuss some of these complications below.

- Exception handling at the Java language level immediately introduces very powerful constraints for types. The Java language requires that a method that overrides another may throw an exception only if it was already declared to be thrown. Consider two methods:

```
class S {
  void foo() throws A, B {...}
}
class C  extends S {
  void foo() throws X, Y, Z {...}
}
```

  The requirement in this case is hard to reason about without an exhaustive search. It can be stated as: "for `C.foo` to be a valid overriding of `S.foo`, `X`, `Y` and `Z` have to be subtypes of either `A` or `B`." Consider how this rich constraint would affect our ability to solve the hierarchy complementation problem at the source level. Imagine that `S` is a known class while `C`, `X`, `Y`, and `Z` are phantom classes. If the language allowed us to infer through observation of other code that `C` is a subtype of `S` and that it provides a method "`void foo() throws X, Y, Z`" then in order to generate a complement we would need to satisfy the following: $\mathtt{C} <: \mathtt{S} \Rightarrow \forall t \in \{\mathtt{X},\mathtt{Y},\mathtt{Z}\} : (t <: \mathtt{A} \vee t <: \mathtt{B})$.

  In contrast, the bytecode verifier only ensures a much simpler constraint: that a type declared to be thrown by a method is a subtype of `Throwable`.

- A similar kind of constraint at the Java language level is also produced by the overriding rule for return types. Java (5 and above) allows overriding methods to have a covariant return type. That is, the overriding method can declare to return a subtype of the overridden method's return type. Much as in the case of exceptions, this induces complex constraints, especially when combined with search to examine whether a type can be a subtype of another. Consider the following case:

```
interface S {
  R foo();
}
// R,X,Y phantom types
// we know X contains method "Y foo()"
```

  For phantom types `R` and `X`, if some other constraint (e.g., of the kind induced in the case of multiple inheritance in Section 3) can be satisfied by making `X` a sybtype of `S`, then we get the additional constraint: "if `X` becomes a subtype of `S` then `Y` must be a subtype of `R`". This is again a very expressive constraint kind and, consequently, hard to reason about. For instance, the above constraint allows us to determine that two phantom types cannot be subtype-related. If two types declare methods with identical argument types but guaranteed-incompatible return types (e.g., `void` and `Object`), then the types are guaranteed to not be ordered by the subtyping relation, in either direction.

  At the bytecode level, subtyping together with signature conformance does not imply other subtyping relationships, in the above manner. By merely having a method with the same argument types, we are not guaranteed that it overrides the respective superclass method. Instead, overloading is perfectly legal among methods that differ only in their return types. The bytecode method call resolution procedure does not rely on name/type lookup but on direct identifiers of methods.

- Casts yield no constraints at the bytecode level although they do at the source level. The reason is that the bytecode elides all unnecessary casts (i.e., upcasts). For instance, at the source level, upon seeing in code that passes the type checker a statement of the form "`(X) new C()`" we can be certain that (assuming `X` and `C` are both classes) the classes `X` and `C` are subtype-related: the cast can be either an upcast or a downcast, otherwise it would fail statically. At the bytecode level, however, a corresponding "`checkcast X`" instruction, when the object at the top of the argument stack is of static type `C`, allows no inference. The corresponding source code could well have been "`(X)((Object) c)`", with the intervening upcast elided during compilation to bytecode.

- Constraints can be induced not just by varying the requirements for the output but also by varying the assumptions for the input. In our setting, we only assumed that the input is legal Java bytecode when complemented with some extra definitions. This is distinctly different from assuming that the input has been produced by the translation of Java source code. (Bytecode could well have been produced via compilers for other high-level languages or via bytecode generators.) For instance, the Java language maintains types for all local variables. At the source level, if we call methods on the outcome of a conditional expression, we are guaranteed to be able to assign a type to it. Consider:

```
A a;
B b;
x = (foo()? a : b);
x.meth();  // I::meth()
x.meth2(); // J::meth2()
```

  In Java source, the above code means that there exists some type `X` (the type given to variable `x`) such that `X` is a subtype of `I` and `X` is a subtype of `J`, while also `A` and `B` are subtypes of `X`. An equivalent conditional in bytecode form does not need to assign a type to `X`. The constraints will be merely: `A` and `B` are subtypes of both `I` and `J`, without allowing us to infer the existence of such an unknown type `X`. Our constraint solving process is significantly simplified by the fact that we never need to infer the existence of more types.

The above is just a sampling of complications that arise if the hierarchy complementation problem is transposed to other domains, requiring the satisfaction of different static requirements. The effectiveness and efficiency of our approach is largely due to the simplicity of the Java bytecode verification requirements. However, other domains give rise to challenging problems, with a wealth of different constraints, possibly appropriate for future work.

8. Related Work

The hierarchy complementation problem is in principle new, although indirectly related to various other pieces of work in the literature.

From a theory standpoint, our problem is an attempt to more fully determine the structure of a partially ordered set. There is no exact counterpart of our algorithms in the literature. However, there has been work on sorting a poset, i.e., completely determining the partial order [8]. The challenge in such algorithmic research, however, is to perform the sorting with a minimal number of queries. None of the interesting devices of our algorithms are present. Specifically, the device of the single inheritance case (if a node can reach two others, they have to be ordered relative to each other) does not apply, and neither does the interesting constraint of the multiple inheritance case (we cannot add direct supertypes to a known node).

Complementing a program so that the result respects static properties is analogous to analyzing only parts of a program but giving guarantees on the result. There are few examples of program analyses of this nature. Notably, Lhotak et al. recently introduced a technique [1] for analyzing an application separately from a library, while keeping enough information (from the library analysis) to guarantee that the application-level call-graph is correct. Furthermore, the Averroes system [2] uses the assumption that the missing code is independently developed in order to produce a worst-case skeletal library. That is, Averroes takes an existing library and strips away the implementation, keeping only the interface between the library and the application. The implementation is replaced with code (at the bytecode level) that performs worst-case actions on the arguments passed into the library, for the purposes of call-graph construction (i.e., the generated code calls all methods the eliminated original code could ever call). Averroes is related to JPhantom but at rather opposite ends of the spectrum: Averroes produces worst-case skeletal implementations, while JPhantom produces minimal, best-case implementations that still respect well-formedness at the type level. At the same time, Averroes assumes the library interface is available and just tries to avoid analyzing library implementations, while JPhantom applies precisely when the library is completely missing. Thus, JPhantom truly applies to the case of partial programs, whereas Averroes analyzes a partial program but under the assumption that the whole program was available to begin with. It would be interesting to treat a partial program first with JPhantom and then apply Averroes to the JPhantom-produced program complement to obtain the worst-case behavior of a plausible interface for the missing code.

Our hierarchy complementation problem bears a superficial resemblance to the *principal typings* problem [3–5]. The principal typings problem consists of computing types for a Java module in complete isolation from every other module it references. I.e., principal typings aim to achieve a more aggressive form of separate compilation, by computing the minimal type information on other classes that a class needs in order to typecheck and compile. Thus, the motivation is fairly similar to ours, but the technical problem is quite different. First, in our setting we already have the result of compilation in the form of bytecode, and bytecode only. Second, our emphasis is on satisfying constraints instead of capturing them as a rich type. Finally, our constraints are of a very different nature from any in the principal typings literature. As discussed in Section 7, the input and output language assumptions crucially determine the essence of an incomplete program analysis problem.

9. Conclusions

We introduced the class hierarchy complementation problem. The problem consists of finding definitions to complement an existing partial class hierarchy together with extra subtyping constraints, so that the resulting hierarchy satisfies all constraints. In the context of Java bytecode and the constraints of the bytecode verifier, the class hierarchy complementation problem is the core challenge of complementing partial programs soundly, i.e., so that they pass the checks of the verifier when loaded together with the generated complement. We offered algorithms for the hierarchy complementation problem in both the single and the multiple inheritance setting, and linked it to practice with our JPhantom bytecode complementation tool.

We believe that the hierarchy complementation problem is fundamental and is likely to arise in different settings in the future, hopefully aided by our modeling of the problem and some of its solution avenues.

Acknowledgments

We gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant; and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award. The anonymous reviewers offered several useful comments that improved the paper. Eric Bodden and Ondřej Lhoták gave valuable early feedback on the paper's high-level idea.

References

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *Proc. of the 26th European Conf. on Object-Oriented*

Programming, ECOOP '12, pages 688–712. Springer, 2012.

[2] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *Proc. of the 27th European Conf. on Object-Oriented Programming*, ECOOP '13, pages 378–400. Springer, 2013.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: compositional compilation for Java-like languages. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '05, pages 26–37, New York, NY, USA, 2005. ACM.

[4] D. Ancona, F. Damiani, S. Drossopoulou, E. Zucca, and D. U. D. Genova. Even more principal typings for Java-like languages. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, 2004.

[5] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '04, pages 306–317, New York, NY, USA, 2004. ACM.

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[8] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and selection in posets. In *Proc. of the 20th Annual ACM-SIAM Symp. on Discrete Algorithms*, SODA '09, pages 392–401, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[9] D. Dig. A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22, Jan. 2011.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java[TM] Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[11] S. Guarnieri and B. Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proc. of the 18th USENIX Security Symposium*, SSYM '09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[12] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13. ACM, 2013.

[14] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[15] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of Javascript applications in the presence of frameworks and libraries. Technical Report MSR-TR-2012-66, Microsoft Research, July 2012.

[16] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '06, pages 308–319. ACM, 2006.

[17] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[18] R. F. Stärk and J. Schmid. The problem of bytecode verification in current implementations of the JVM. Technical report, ETH Zürich, 2000.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[20] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proc. of the 9th Int. Conf. on Compiler Construction*, CC '00, pages 18–34, 2000.

A. Multiple Inheritance Correctness Proof

We shall call the path-edges originating from known-nodes kp-*edges*. We will also use the symbols $S_0, S_1, \ldots, S_\infty$ to denote the various stratifications computed at each step of Algorithm 3.1. Note that our algorithm will actually produce a finite number of stratifications (at most $|V|$) but we can disregard both the upper limit of the main loop and the early-failure condition (line 23) for proving correctness. Instead we focus on the main computation (line 18) and the infinite sequence of stratifications that would be produced if it was allowed to run forever (even after reaching a fixpoint).

Lemma 1. *For all $v \in V$, the sequence $\{S_0[v], S_1[v], \ldots\}$ is non-decreasing.*

Proof. Direct consequence of line 18 of the algorithm. □

Lemma 2. *For all $i \in \mathbb{N}$, $0 \le S_i[v] \le i, \forall v \in V$.*

Proof. Induction on step i.

1. *Base*: For $i = 0$, we have that $S_i[v] = S_0[v] = 0, \forall v \in V$.
2. *Inductive Step*: Assume that $0 \le S_n[v] \le n, \forall v \in V$ for some value of n. We must show that $0 \le S_{n+1}[v] \le n{+}1, \forall v \in V$. Let k be a node in V. Either $S_{n+1}[k] = S_n[k]$, and therefore $0 \le S_{n+1}[k] \le n$, or there will exist a node s, s.t. $S_{n+1}[k] = S_n[s] + 1$, in which case $1 \le S_{n+1}[k] \le n + 1$.

□

Definition A.1. A node $v \in V$ is *i-stabilized* if and only if $S_i[v] = S_{i+1}[v]$ and either $i = 0$ or $S_{i-1}[v] < S_i[v]$.

Theorem 1. *(Once a node's stratum does not change, it will not change again.) If* $S_i[v] = S_{i+1}[v]$ *for some node* $v \in V$ *and a value* $i \in \mathbb{N}$*, then* $S_j[v] = S_i[v], \forall j \in \mathbb{N}$ *such that* $j \geq i$*.*

Proof. Induction on step i.

1. *Base*: For $i = 0$, let v be a node in V, such that $S_0[v] = S_1[v]$. From Lemma 2, we have that $S_0[v] = S_1[v] = 0$, which can happen if and only if v has no incoming edges (otherwise an edge would cause the node to move to a higher stratum on iteration 1). It is therefore impossible for v to change in the following iterations since it has no constraining edges.
2. *Inductive Step*: Assume that the theorem holds for all $i < n$ for some value of $n \in \mathbb{N}$. Let $t \in V$ be a node, such that $S_n[t] = S_{n+1}[t]$. We will show that t's stratum will not change in the future. It suffices to prove that, for each of t's constraining edges, there will be a node s that has already been stabilized at a lower stratum than t, and can be used to satisfy the constraint at this point. Therefore, the constraint will remain satisfied in future iterations due to s, which will remain in the same stratum from now on (induction hypothesis). For ordinary *path*-edges, node s is no other than the source of the edge itself, while for *kp*-edges, it is the lower phantom projection of the edge's source at step n that we may use instead. Let us consider ordinary path-edges first, in more detail. From Lemma 2, we have that $0 \leq S_n[t] \leq n$, and thus $0 \leq S_{n+1}[t] \leq n$. Let $(s,t) \in E$ be an incoming edge of t. We have that $0 \leq S_n[s] < S_{n+1}[t] \leq n$ which entails that $0 \leq S_n[s] \leq n-1$. Therefore, according to Lemma 2, we have that $0 \leq S_i[s] \leq n-1, \forall i \in \{0,\ldots,n\}$. By the pigeonhole principle, and due to Lemma 1, there must surely exist two consecutive values $i, i+1$, s.t. $S_i[s] = S_{i+1}[s]$ and $i < n$. From the induction hypothesis, we know that s will therefore not change and its constraint on t will be irrelevant in future iterations. We proceed similarly, for a *kp*-edge (s,t) (where we use the lowest phantom projection of s at this point, instead of s itself). Thus, t will not change in the future, since every constraint of t will remain satisfied after this iteration.

□

Corollary 1. *For all* $v \in V$ *and* $n \in \mathbb{N}^+$, $S_{n-1}[v] \neq S_n[v] \Rightarrow S_n[v] = n$.

Theorem 2. *The stratification sequence* $S_0, S_1, \ldots$ *will diverge (i.e., not reach a fixpoint) if and only if at some computation step, n, no new nodes stabilize and not all nodes have already stabilized—that is,* $\exists n \in \mathbb{N}^+$*, such that: for some* $v \in V$, $S_{n+1}[v] \neq S_n[v]$ *and for all* $v \in V$, $S_{n+1}[v] = S_n[v] \Rightarrow S_n[v] = S_{n-1}[v]$.

Proof.

1. (*If*) Let n be a computation step, such that $(\forall v \in V)$ $(S_{n-1}[v] \neq S_n[v] \Rightarrow S_n[v] \neq S_{n+1}[v])$. We can disregard any node u such that $S_{n-1}[u] = S_n[u]$, and observe that for each remaining node, there must exist at least a constraining edge that cannot be satisfied with a node that has already been "stabilized". That said, due to Corollary 1, each remaining node $v \in V$, s.t. $S_{n-1}[v] \neq S_n[v]$, will be placed at a higher (by 1) stratum at this point, i.e., $S_i[v] = i, \forall i \in \{0,\ldots,n+1\}$. Since the relative positions of all the remaining nodes will be the same at step $n+1$ as they had been at step n, there is no way for a node to be stabilized at this last iteration. In other words, there is a cyclic dependency between the remaining nodes that will remain unaltered, thus eliminating the possibility of reaching a fixpoint.
2. (*Only If*) Due to Theorem 1, we know that we need at most $|V|$ computation steps, in order to reach a fixpoint, if at each computation step there exists at least a new node that gets stabilized. In other words, we need a finite number of steps to reach a fixpoint, if each step results in some progress. Thus, failure to reach a fixpoint requires an iteration where no progress has been made, i.e., no new nodes get stabilized.

□

Therefore, the optimization in Algorithm 3.1 of detecting this exact condition (line 23) and terminating would be triggered if and only if no fixpoint would be reached whatsoever, had the algorithm continued its execution.

Soundness. We need to show that, if our algorithm computes a solution, this solution will be sound. Firstly, our algorithm maintains the invariant that $\forall (s,t) \in E$, node s will eventually—i.e., once we reach a fixpoint—be placed somewhere lower than node t (otherwise this condition would trigger yet another iteration). Therefore, our solution will contain no cycles since all of its edges will be facing *upwards*, i.e., from a lower to a higher stratum. Furthermore, it is evident that, for each *kp*-edge (s,t), there will always exist a node $p \in proj(s)$, such that p will be placed at a lower stratum than t in our final solution. Thus, we can add the edge (p,t) without introducing any cycles if none existed so far. This process will therefore generate a valid solution. □

Completeness. We need to show that, if a solution exists for a given constraint graph, then our algorithm will also be able to compute a solution, or equivalently (according to Theorem 2) that the stratification sequence being computed will reach a fixpoint. Consider such a (posited but unknown) solution. For such a solution we may generate a stratification (since the solution may contain no cycles), such that each of its edges is facing upwards and no empty strata exist, that is, $\forall (s,t) \in E_{sol} : S_{sol}[s] < S_{sol}[t]$, where E_{sol} are the edges that form the solution, and S_{sol} is its stratification. We first show an important lemma.

Lemma 3. *Let S_{sol} denote the stratification of a valid solution of the problem instance. We have that: $\forall i \in \mathbb{N}, \forall v \in V : S_i[v] \leq S_{sol}[v]$.*

Proof. Suppose that there is a step $k \in \mathbb{N}$, such that it contains at least one node $u \in V$ with $S_k[u] > S_{sol}[u]$, and without loss of generality, suppose that k is the smallest such integer, i.e., that before that point our stratification was upper bounded by that of the unknown solution: $\forall j \in \{n \in \mathbb{N} \mid 0 \leq n < k\}, \forall v \in V : S_j[v] \leq S_{sol}[v]$. For u to be placed at a higher stratum by our algorithm there must exist an edge $(s, u) \in E$ such that either (i) (s, u) was a constraining ordinary path-edge: $S_k[u] = S_{k-1}[s] + 1$, or (ii) (s, u) was a *kp*-edge and $\forall p \in proj(s) : S_k[u] = S_{k-1}[p] + 1$. In the first case, we have that: $S_{sol}[u] \leq S_{k-1}[s] \leq S_{sol}[s]$, and since (s, u) must also be present in the solution, this violates the single-direction edge property. In the second case, $S_{sol}[u] \leq S_{k-1}[p] \leq S_{sol}[p], \forall p \in proj(s)$, which leads to another contradiction, since there must exist a node $p \in proj(s)$, such that a path exists from p to u in the solution, which can only happen if p was placed at a strictly lower stratum than u. Thus, since all possible cases lead to a contradiction, we have proved our initial proposition: our algorithm always assigns to every node a stratum that is lower than, or equal to, that of any true solution of the problem instance. □

Let $s_{sol} = \sum_{v \in V} S_{sol}[v]$ and $s_i = \sum_{v \in V} S_i[v], \forall i \in \mathbb{N}$. It follows that $s_i \leq s_{sol}, \forall i \in \mathbb{N}$, for any such possible solution. Additionally, because of Lemma 1 and our two theorems, we know that each step but the last will strictly increase the sum of all strata values over all nodes. E.g., if our algorithm ended its execution at step n, then we would have: $s_0 < s_1 < \ldots < s_{n-1} = s_n$.

Suppose there is a solution but our algorithm fails to compute one (i.e., no fixpoint will ever be reached). Since s_i strictly increases at each step of our algorithm, and the only way to return is by finding a valid solution, we know that there will exist a step n, such that $s_n > s_{sol}$. However, this contradicts our proven proposition that $s_i \leq s_{sol}, \forall i \in \mathbb{N}$. Therefore, we conclude that if valid a solution exists, our algorithm will also be able to compute one.

□

Theorem 3 (Principality). *Any solution produced by Algorithm 3.1 will have a minimum number of strata. That is, for any possible solution of the problem instance, with t denoting the solution's total strata, we have that $n \leq t$, where n is the total number of strata produced by Algorithm 3.1.*

Proof. Let S_{sol} denote the stratification of a valid solution of the problem instance, and t its total number of strata. Without loss of generality we assume that strata are denoted as consecutive integers starting from 0, beginning from the lowest stratum. Thus, $\forall v \in V : S_{sol}[v] < t$.

Since a solution exists and *completeness* has been proved, we know that Algorithm 3.1 will also be able to terminate successfully at some step $n \in \mathbb{N}$, yielding its own solution. Let n_s be the total number of strata, and $x \in V$ be a node at the highest stratum of the solution computed by our algorithm. That is, $\forall v \in V : S_n[v] \leq S_n[x]$. Node x will also be the node that was changed last, which is at step $n - 1$, i.e., $S_n[x] = S_{n-1}[x] \neq S_{n-2}[x]$. Therefore, from Corollary 1, we have: $S_n[x] = S_{n-1}[x] = n - 1$. Since strata are consecutive integers starting from 0, we have that: $n_s = S_n[x] + 1 = n$.

According to Lemma 3, we have: $n = n_s = S_n[x] + 1 \leq S_{sol}[x] + 1 \leq t < t + 1$. Thus, we have shown that our algorithm minimizes the total number of strata it produces. □