KABA

A System for Refactoring Java Programs

Mirko Streckenbach

Januar 2005

Dissertation

Eingereicht an der

Fakultät für Mathematik und Informatik

der

Universität Passau

Acknowledgments

First of all, I wish to thank my adviser Gregor Snelting for giving me the opportunity to do this work without any pressure. I also want to thank the second reviewer, Barbara Ryder.

I thank my parents for giving me a good education and enabling my to study computer science.

Special thanks go to two students who contributed to KABA: Peter Schneider, who created a bytecode transformer for KABA, and Thorsten Buckley, who created an alternate implementation of KABA's analysis.

I also want to thank my colleagues in Passau: Jens Krinke, Torsten Robschink, Maximilian Störzer and Christian Hammer.

Steven Milosevic helped to improve the English of this thesis. A 'big thanks' goes to him, too.

This work was funded by Deutsche Forschungsgemeinschaft, grants DFG Sn11/7-1 and Sn11/7-2.

This thesis is dedicated to Peter Ulrich Biehl, who has been a invaluable friend for more than 20 years. His sudden death was a big loss.

Abstract

Refactoring is a well known technique to enhance various aspects of an object-oriented program. It has become very popular during recent years, as it allows to overcome deficits present in many programs.

Doing refactoring by hand is almost impossible due to the size and complexity of modern software systems. Automated tools provide support for the application of refactorings, but do not give hints, which refactorings to apply and why. The Snelting/Tip analysis is a program analysis, which creates a refactoring proposal for a class hierarchy by analyzing how class members are used inside a program.

KABA is an adaption and extension of the Snelting/Tip analysis for Java. It has been implemented and expanded to become a semantic preserving, interactive refactoring system. Case studies of real world programs will show the usefulness of the system and its practical value.

Contents

1 Introduction **11**
1.1 Refactoring 11
1.2 The Snelting/Tip Analysis 12
1.3 Preservation of Semantics 16
1.4 Other Work 16
1.4.1 Bowdidge and Griswold 16
1.4.2 Casais 17
1.4.3 Kataoka, Ernst, Griswold, and Notkin 17
1.4.4 Moore 18
1.4.5 Rajesh and Janakiram 18
1.4.6 Tip, Kiezun, and Bäumer 19
1.5 Overview 19
1.6 Accomplishments 19
1.7 Pseudo-code Notation 20

2 The Snelting/Tip Analysis **21**
2.1 Concept Analysis 21
2.2 The Original Snelting/Tip Analysis 23
2.2.1 Defining the Table 23
2.2.2 Constructing the Table 25
2.2.3 Constructing the Lattice 29

3 Extensions to Snelting/Tip **33**
3.1 Static Methods and Fields 33
3.2 Class Initializers 34
3.3 Runtime Type Checks 37
3.4 Type-Casts 40
3.5 Exception Handling 41
3.6 Equality of Signatures for Overridden Methods 42
3.7 Multiple Implementations in the Same Class 45
3.8 Library Classes 49
3.8.1 Reducing Table Entries for Library Classes 50
3.8.2 Recreation of Library Classes 54

3.8.3 Type-Constraints generated by Library Classes 55

4 Points-To Analysis 57
4.1 Background 57
4.2 Requirements 58
4.3 Andersen for Java 59
4.4 Steensgaard for Java 62
4.5 Flow-Sensitivity 62
4.6 Context-Sensitivity 65
4.7 Object-Sensitivity 69
4.8 Other Points-To Analyzes for Java 71

5 An Analysis of Java-Bytecode 73
5.1 Introduction 73
5.2 Intra-Procedural Analysis 77
5.3 Inter-Procedural Analysis 83
5.4 Whole Program Analysis 83
5.4.1 Stubs 84
5.4.2 A Conservative Approximation of Unanalyzed Code . 85
5.4.3 Smart Stubs 86
5.5 Snelting/Tip for this Analysis 87
5.6 Reflection 88
5.7 Contexts 90
5.8 Differences to a Real Java Virtual Machine 90
5.9 Implementation 90

6 Dynamic Snelting/Tip 93
6.1 Creating the Table at Run-Time 93
6.2 Limitations 95
6.3 Implementation 96

7 Semantic Preserving Refactoring 99
7.1 Class Hierarchy for a Concept Lattice 100
7.2 The Lookup Algorithm 101
7.3 Preserving Semantics for Modifications 102
7.4 Efficient Check for Broken Semantics 106
7.4.1 Caching Semantic Information 106
7.4.2 Creating Incremental Caches 107
7.4.3 Comparing Two Caches 107

8 Refactoring Algorithms 113
8.1 Basic Manipulation of the Class Hierarchy 113
8.2 Creation of Library Classes 115
8.3 Automated Simplification 118

8.4 Removing Multiple Inheritance 123
8.5 Other Algorithms 127
8.5.1 Removing Empty Classes 127
8.5.2 Removing Type-Check Nodes 127
8.5.3 Merging Nodes with only a This-Pointer 128
8.5.4 Merging Nodes Always 131

9 KABA 133
9.1 Overview 133
9.2 The Graphical Refactoring Editor 133
9.3 The KABA Refactoring System 136

10 Case Studies 139
10.1 A “small” example 139
10.2 *javac* 141
10.2.1 The Visitor Pattern 141
10.2.2 The Syntax Tree 143
10.2.3 The Symbol Table 143
10.3 *antlr* 146
10.3.1 Syntax Tree 146
10.3.2 Token 148
10.4 Varying the Number of Clients 151
10.5 Performance 153

11 Discussion 157

12 Future Work 159

A Constraints for the Bytecode Analysis 161

B Limitations of the Prototype 231
B.1 Late Addition of Assignments 231

Chapter 1

Introduction

Design of a class hierarchy is difficult. The designer must try to foresee all possible uses for a particular class. As nowadays class hierarchies often consist of hundreds of classes this job has become complex due to the pure size of the hierarchy. Evolution of the class hierarchy even increases this problem. New functionality must be added, new concepts must be integrated and entropy is often increased. During an evolutionary process, new maintainers take over, who are often not familiar with the spirit behind the original design and leave a different "handwriting" in their contribution to the class hierarchy. Of course, in many cases the clients using this library shall not be affected by all those changes.

The class provided by the Java Development Kit are a prime example of this: From 1.0 to 1.5 the number of classes in the `java.` namespace has grown almost by a factor of 10 (from 235 to 2158). Version 1.5 provides two different approaches to GUI programing, 3 different approaches to I/O, and 4 different approaches to container classes. Although some of the old concepts are marked as depreciated or obsolete, they are still in wide-spread use.

1.1 Refactoring

An established technique against mistakes in design or the negative effects of evolution is *refactoring*. Refactoring is a generic term used to express that a program is transformed in order to improve its structure, readability or maintainability. It was invented by Opdyke and Johnson [33, 32], but the term became most widely known after Fowler's book [13] which presents a whole catalog of *refactoring patterns* that can be applied to a program.

While refactoring may look like a manual process at first, it is quite obvious that this is not true for large programs. Renaming a local variable in a ten line program may be done fast with any editor, but renaming a field of a class that is used hundred of times in a 100000 line program cannot.

A lot of modern programing environments (e.g. Eclipse[1]) already have support for application of Fowler's and other refactorings. But most provide support only for the application of these refactorings and do not make suggestions which refactorings should be applied. Automatic creation of refactoring proposals is still in its infancy.

There are different reasons for that. First, most refactorings come with non trivial preconditions, which can often only be checked by complex program analysis. Secondly, for many refactorings, a "counter"-refactoring exists. The decision, whether the refactoring shall be applied or not may be non trivial for a human and is even more difficult for a machine.

1.2 The Snelting/Tip Analysis

The Snelting/Tip analysis [43] suggests a program transformation, which analyzes a class hierarchy and its clients and creates a refactoring proposal in form of a new class hierarchy. This hierarchy is specific to the analyzed clients. For them, it is semantically equivalent to the original program and minimal in the sense that every object and pointer contains only the members it needs. The semantical equivalence is reached by combining program analysis, type constraints and concept lattices.

Figure 1.1 shows a small class hierarchy and two example clients. It models humans in a university. A class `Person` has data-members for name, address and social security number of a person. This class is extended by the class `Student`. A student additionally has a student id and a professor maybe his advisor. A second subclass `Professor` has a work address and a may have a student as his assistant. Then there are two clients that use the hierarchy. In both clients a student and a professor object is created. In the first client, the professor is made the advisor of the student, while in the other client, the professor hires the student as his assistant.

The hierarchy suggested by Snelting/Tip can be seen in section 1.2. Every class is displayed as a horizontally split box, where the upper half contains members of the class and the lower half objects or pointers whose type is that class. Pointers are named after their variable name in source code and objects are given the name of their type and an additional number to make different objects distinguishable. There are several things to notice:

- There is a class above `java.lang.Object`. It has no members, but a lot of pointers. This indicates these pointers do not access any members. Pointers appearing there are good candidates for further inspection, as unused variables are often signs of programming errors or redundant or erroneous declarations. This applies to pointers like `p1` as well as

[1] `www.eclipse.org`

```
1   class Person {
2      public String name;
3      public String address;
4      public int socialSecurityNumber;
5   }
6
7   class Student extends Person {
8      public int studentId;
9      public Professor advisor;
10     public Student(String sn, String sa, int si) {
11        name = sn;
12        address = sa;
13        studentId = si;
14     }
15     public void setAdvisor(Professor p) {
16        advisor = p;
17     }
18  }
19
20  class Professor extends Person {
21     String workAddress;
22     Student assistant;
23     public Professor(String n, String wa) {
24        name = n;
25        workAddress = wa;
26     }
27     public void hireAssistant(Student s) {
28        assistant = s;
29     }
30  }
31
32  class Client1 {
33     static public void main(String args[]) {
34        Student s1 = new Student("Carl", "here", 12345678);
35        Professor p1 = new Professor("Prof. X", "there");
36        s1.setAdvisor(p1);
37     }
38  }
39
40  class Client2 {
41     static public void main(String args[]) {
42        Student s2 = new Student("Mary", "here too", 87654321);
43        Professor p2 = new Professor("Prof. Y", "not there");
44        p2.hireAssistant(s2);
45     }
46  }
```

Figure 1.1: Example: professors and students

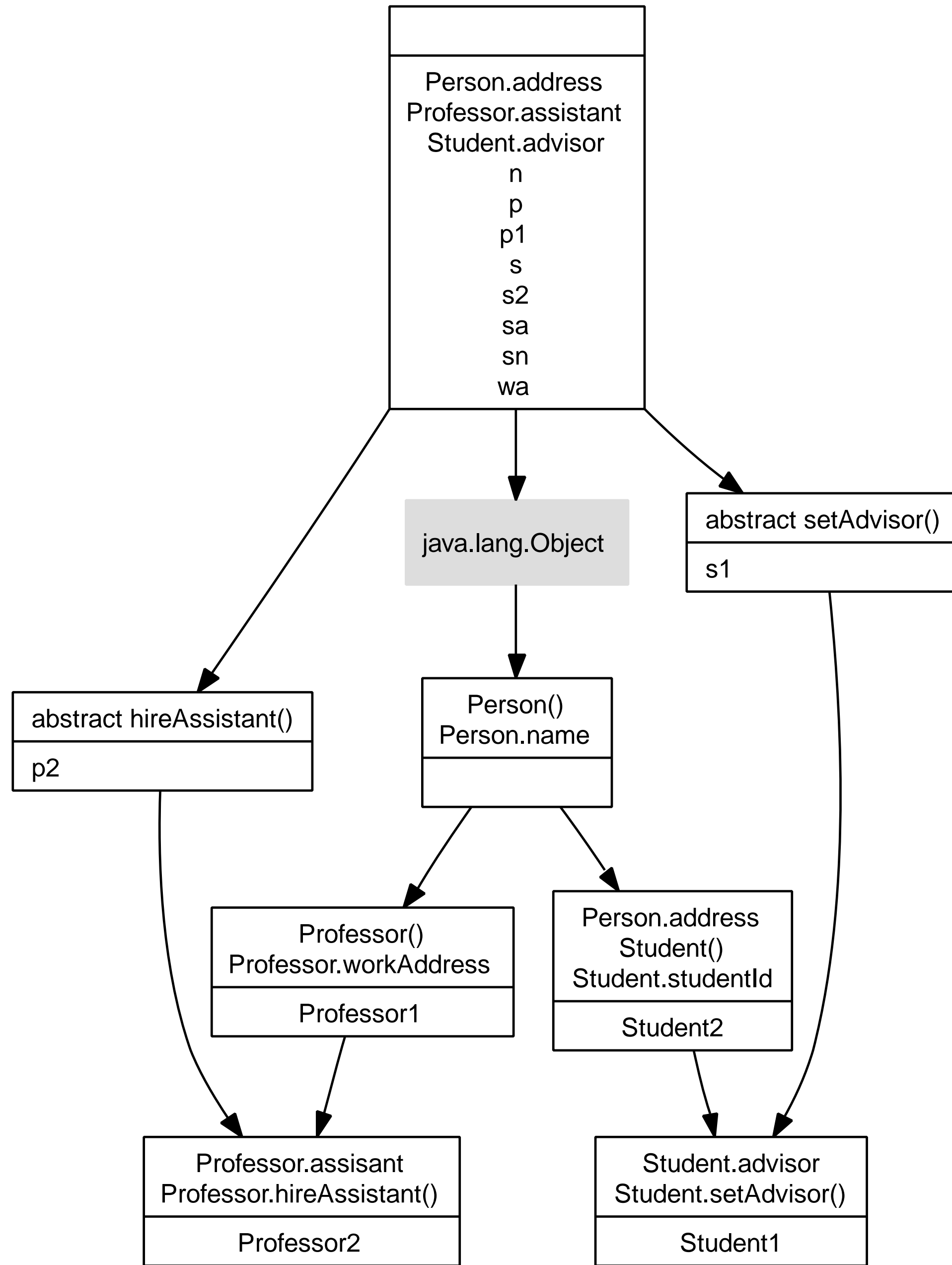


Figure 1.2: Refactoring proposal for figure 1.1

field declarations like `Person.address`[2].

- Members of the class `Professor` appear at two different classes. The class for the object `Professor1` contains only the field `workAddress` and the constructor. The field `assistant` and the method `hireAssistant` have been moved into a new subclass and the object `Professor2` has this subclass as its type. The analysis detects that there are two different kinds of professors: one without and one with an assistant and thus suggests creating two different classes for them.

- Something similar happens for the original student class: it is split into two classes for students with or without an advisor.

- The field `address` has been moved from its original class `Person` into the class `Student`. This was done because no professor uses this field.

- For the pointer `p2`, creation of a special interface containing only a declaration of the `hireAssistant` method is suggested.

- Similarly for the pointer `s1`, creation of an interface containing only `setAdvisor` is suggested.

- The member `socialSecurityNumber` has been removed from the hierarchy entirely, as it was unused.

It is important to view this hierarchy as a refactoring proposal. It should be reviewed by a programmer who is familiar with the code. He can then make further modifications to the class hierarchy, e.g. merge two classes if their distinction is not required or reanimate dead members if they are known to be needed in future applications. Many other known refactorings like *move method* or *move attribute* can be applied too. Other refactorings applied by the analysis like *extract interface* can be undone if their effect is unwanted.

The initial proposal or the manually manipulated hierarchy can then be used in various way. First of all, as it is a refactoring proposal, it may be used to automatically transform the original program into this new hierarchy. As this is a very invasive transformation to the program source, some developers may not want it. Still, the new hierarchy can give them interesting insights into the actual use of members and attributes within the analyzed program. This may be a starting point for manual applications of refactorings.

[2]The huge amount of pointers in this is example is an artifact of the small size of the clients.

1.3 Preservation of Semantics

The term *semantics-preserving* and *semantically equivalent* will be used within this thesis. It is hard to specify exactly what is meant by this, as almost every transformation done to a Java program may result in a change of behavior - if it is not visible in terms of program output, it may be measurable in terms of program performance. Snelting/Tip basically preserves the access to members. All objects will access the same members as before, regardless of the type of access used: static bound access to data members and dynamic bound access to methods during run-time. It also preserves behavior of objects with regards to dynamically checked properties based on types (i.e. `instanceof` and type-casts). Problematic are programs which make more in-depth assumptions about the structure of the class hierarchy, e.g. the name of a class or the number of methods in a class. In addition, changing the class hierarchy may result in different timings while loading classes and thus change observable program behavior, but this kind of behavior is not guaranteed by either the Java language or the virtual machine and similar changes could also be observed by switching to a different virtual machine.

1.4 Other Work

1.4.1 Bowdidge and Griswold

Bowdidge and Griswold [6] created a program visualization called *star diagram* that visualizes the access to data structures within a program. For a programmer selected component of a program, the star diagram displays the elements of the program that contains or references the selected component. This visualization allows the application of restructurings like extraction or inlining of functions and their parameters. Star diagrams are generally helpful for encapsulation. They provide generators of star diagrams for C, but their tool was originally based on Scheme.

This approach differs from Snelting/Tip in a number of ways. First, their focus are not object-oriented programs, but other languages, so their list of restructurings omits all problems specific to object-orientation. Secondly, their approach is statement based and Snelting/Tip cannot suggest refactorings like *extract function*. Their approach works on a subset of all refactorings available disjunct to the set that Snelting/Tip will propose. And thirdly, their tool only enables the application of restructurings, it will not make proposals.

1.4.2 Casais

Casais[8] presents an algorithm that can be used to restructure a class hierarchy at the introduction of a new class. In some cases, a programmer requires only a part of the members already present in a class hierarchy, but all existing classes have additional *unwanted* members. The algorithm restructures the existing hierarchy in a way, that a class containing exactly the set of members needed will be present in the new hierarchy.

The obvious difference to Snelting/Tip is the fact that Casais' algorithm is incremental. On the other hand, both algorithms have in common that the class hierarchy is minimal after their application. In order to use Casais' algorithm, the set of members that shall be contained in the new class must be specified. Snelting/Tip will infer this set automatically from the given program. Instead of specifying the members, a programmer could just choose any exiting class that fits his needs, use only the members required, and than run Snelting/Tip and he should get a result that is similar to the result of Casais' algorithm.

Casais' does not give any semantical guarantees and as his algorithm may introduce multiple inheritance (the algorithm was written with Eiffel in mind), this is a serious problem, because member access may become ambiguous and this ambiguity has to be resolved by hand.

1.4.3 Kataoka, Ernst, Griswold, and Notkin

This work[21] presents an approach to automatically generate refactoring proposals. These proposals are based on automatically detected invariants. E.g. if a method that calculates the area of a rectangle and has *width* and *height* as parameters is only used for squares, an invariant *width* = *height* can be detected. Based on that, the refactoring *remove parameter* would be proposed for application. Their invariant detection tool Daikon[31] can of course also detect more complex invariants, but it is a tool that is based on execution of the program, not on static analysis. The suggestions however, can be based on any kinds of invariants, regardless whether they are detected by static or dynamic analysis or given manually.

There are a number of differences between this work and Snelting/Tip. First of all, the set of refactorings they can proposed are disjunct from the set of refactorings used by Snelting/Tip, as their refactorings are applied at the statement level and thus are more local. The main source used for the invariants, on which their refactoring proposals are based, is a dynamic analysis, whereas Snelting/Tip is usually based on a static analysis. The proposal system supports statically generated invariants, but such invariants are much harder to detect automatically and if those actually can be used to create useful refactoring proposals remains open. In addition to that, it is not clear from the paper, if their system provides enough information to

actually transform the code or generates only the refactoring proposal and then relies on other tools for application of the refactoring. Both analyzes have in common, that the refactoring proposal is based on the actual usage of variables within a program and therefore both are client specific.

1.4.4 Moore

Moore [28, 29, 30] describes a tool called Guru, which does automatic class hierarchy restructuring and method refactoring for programs written in Self. It restructures method in a class hierarchy in an optimal way, so that no method appears twice in a hierarchy. Removing duplicate methods is possible in Self, because equivalence of two methods can be detected if their source code is equivalent. Additionally, he restructures methods by creating new methods for common expressions.

Self is not a class centric language like Java. Here, the term class is meant to refer to both, classes and objects. If only looking at the class hierarchy restructuring, this approach may have a lot of similarities to Snelting/Tip at first glance, but there are also a lot of differences. First of all, his class hierarchy restructuring is based only on the set of members for a class. Whether these members are used or not is not taken into account, nor is information about the position of members in the previous hierarchy (like Snelting/Tip's hiding constraints). The algorithm he uses to create a new hierarchy also creates a minimal hierarchy. The difference to the concept analysis based approach are additional suprema and infima in a lattice. Additionally, there are a lot of small differences caused by the fact that Java and Self are languages on different ends of the object-oriented spectrum. Despite all theses differences, this is the only other algorithm known, which restructures existing class members into a whole new hierarchy.

1.4.5 Rajesh and Janakiram

An approach for the automatic creation of refactoring proposals was done by Rajesh and Janakiram [34]. Their approach is specialized in searching refactoring opportunities to introduce design patterns[15] into the program. They transform a given program into *predicate-templates* and then use Prolog rules for the identification of possible sites for the application of refactoring.

The refactorings for design patterns are quite complex, so it should be possible to use their approach to find possible applications for more low-level refactorings like *move member*. As their tool makes only suggestions and does not transform the code, the questions whether applying a suggested refactoring will preserve program semantics is left open.

1.4.6 Tip, Kiezun, and Bäumer

In [50] a system which helps to apply refactorings is presented. The refactoring has to be selected and parametrized by the a user and is then automatically applied to the Java source code. Type constraints are used to infer which types in the source code can and have to be changed, if the application of the refactoring is meant not to break program semantics. Their focus is on generalizing refactorings (i.e.g *extract interface*), but the same mechanism can be used for other refactorings, too.

The main difference to Snelting/Tip is that they do not make refactoring proposals, but focus on their application. Both methods complement each other: If the hierarchy suggested by Snelting/Tip cannot be used as a whole, it can still be used to get candidates for interface extraction, which then can be executed using their algorithms.

1.5 Overview

This thesis is structured into four parts. First, a short recapitulation of concept analysis and the Snelting/Tip analysis is given, then Snelting/Tip is adapted from C++ to Java. The second part deals with the implementation of the analysis. The use of points-to analysis for Snelting/Tip is examined and a constraint based system for the extraction of information from Java bytecode is presented. Additionally, a dynamic version of the Snelting/Tip analysis is given, based on virtual machine instrumentation instead of static program analysis. The third part focuses on interactive refactoring. A semantic model for behavior preserving refactoring is presented, along with some algorithms which enhance the structure of a class hierarchy. Finally the prototype implementation is described shortly and case studies are presented.

1.6 Accomplishments

The accomplishments of this thesis are:

- Adaption of the Snelting/Tip analysis from a subset of C++ to full Java.
- An analysis of stack-based Java bytecode which extracts the input needed by the Snelting/Tip analysis.
- A new dynamic version of the Snelting/Tip, which collects access information at program run-time and solves possible scalability issues of the traditional static analysis.

- A semantic model that allows fast checking if a modification to the class hierarchy breaks program semantics.
- The design of KABA, a tool which allows interactive refactoring of class hierarchies analyzed with Snelting/Tip.
- Case studies which evaluate the usefulness of Snelting/Tip

1.7 Pseudo-code Notation

All algorithms in this thesis are given in a pseudo-code notation. While the notation was meant to be intuitive, some words of explanation might be helpful.

- Blocks ruled by `if` or `while` can be recognized by their indentation level. Any statement with an indentation equal to or smaller than the loop predicate will end the block.
- $|s|$ denotes the number of items in a set s.
- $\exists e$: *condition* is used as a boolean expression and binds e for the current block to a value that satisfies the condition.
- $\forall e \in E$: *condition* is used a loop operator. No particular order for the values of e is assumed.
- $a \equiv b$ is used a boolean operator to test for structural identity. If b contains unbound variables, these are implicitly existential qualified and like the $\exists$ operator bound for the current block.

Chapter 2

The Snelting/Tip Analysis

This chapter gives a short recapitulation of the original analysis presented by Snelting and Tip[42, 43], as well as a short introduction to the required elements of concept analysis. The original analysis was written for C++. The notation and some of the terminology has been adapted to Java and the analysis core had to be changed a bit due to the more subtle difference between C++ and Java. All of these are minor modifications, more severe modifications to the analysis will be presented in chapter 3.

This chapter is not a complete reproduction of the original papers. They provide more examples and discussion about the analysis. For these details, the reader is referred to [43].

2.1 Concept Analysis

Binary relations between two different sets M and N can be displayed by listing all elements (m, n) with $m \in M$ and $n \in N$. In 1940 Garrett Birkhoff created the mathematical foundations of concept analysis. He proved that a lattice can be constructed for each binary relation. The lattice is an alternative representation of the elements of the relation. It can be converted into the original relation and vice versa. The lattice provides a different view on the structure of the original relation and exposes properties that may remain hidden in the original representation.

Later concept analysis was used for the analysis of data by Ganter and Wille[16]. It has been used for many different purposes since then, including applications in the area of software engineering [39, 9, 4, 55, 25, 40, 41, 2, 26].

Concept lattices are created for a binary relation between a set of *objects* $\mathcal{O}$ and their *attributes* $\mathcal{A}$. The relation can be stored in a boolean table $T \subseteq \mathcal{O} \times \mathcal{A}$. For any set of attributes $A \subseteq \mathcal{A}$, the set of objects having all these attributes can be determined by

$$\rho(A) = \{o \in \mathcal{O} | \forall a \in A : (o, a) \in T\}.$$

Accordingly for a set of objects $O \subseteq \mathcal{O}$, the set of attributes used by these objects is defined by

$$\sigma(O) = \{a \in \mathcal{A} | \forall b \in O : (o, a) \in T\}.$$

Then, a concept is a pair (O, A) if

$$\rho(A) = O \wedge \sigma(O) = A.$$

A concept can been seen as a rectangle of table entries in T where the order of rows and columns is irrelevant. Each object $o \in O$ has all attributes $a \in A$ and each attribute $a \in A$ is used by every object $o \in O$.

A relation $\leq$ can be defined between two concepts (O_1, A_1) and (O_2, A_2) as follows:

$$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2 \Leftrightarrow A_1 \supseteq A_2.$$

The set of concepts builds a complete lattice, the *concept lattice* $\mathcal{L}(T)$. For two elements, the infimum is defined as

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

and the supremum as

$$(O_1, A_1) \vee (O_2, A_2) = (\rho(A_1 \cap A_2), A_1 \cap A_2)).$$

All diagrams used for concept lattices use a reduced notation. In this notation, the concept $\gamma(o) = (\rho(\sigma(\{o\})), \sigma(\{o\}))$ is labeled with the object o and the concept $\mu(a) = (\rho(\{a\}), \rho(\sigma(\{a\})))$ with the attribute a.

The relation between the table T and the lattice can be expressed as

$$(o, a) \in T \Leftrightarrow \gamma(o) \leq \mu(a).$$

This also shows, that lattice and table display the same information and are convertible into each other. In the lattice, all attributes of an object o can be found at the concepts *above* o. Similar, all objects which have a certain attribute a, can be found below a. Then infima show attributes which are used by the same objects, while suprema show objects having the same attributes. This hierarchical structure is not visible in the table and exploited by transforming it into a lattice.

The table can be enriched with additional background knowledge in order to influence the positioning of objects and attribute in the lattice. If an attribute b must be above a, an implication can be added:

$$a_1 \rightarrow a_2 \Leftrightarrow \mu(a_1) \leq \mu(a_2) \Leftrightarrow \forall o \in \mathcal{O} : (o, a_1) \in T \Rightarrow (o, a_2) \in T$$

Of course, the same can be done for implications between two objects:

$$o_1 \rightarrow o_2 \Leftrightarrow \gamma(o_1) \leq \gamma(o_2) \Leftrightarrow \forall a \in \mathcal{A} : (o_1, a) \in T \Rightarrow (o_2, a) \in T$$

The lattice can by constructed from the table with an algorithm by Ganther [16]. Complexity of that algorithm is $O(|\mathcal{O}|^2 \times |\mathcal{A}| \times n)$, where n is the number of concepts. Lindig [26] has implemented different algorithms for lattice construction and evaluated them. In general it can be said that exponential complexity is rarely seen in practice and Lindig's implementation, which was used here, can calculate a lattice of 2000 elements in less than a second.

2.2 The Original Snelting/Tip Analysis

2.2.1 Defining the Table

First of all, some terminology must be introduced. In what follows, $\mathcal{P}$ denotes the analyzed program. This may be a single program including a class hierarchy or a class hierarchy and multiple client programs. In all cases the program must be complete and include all code that may be executed. $C.m$ denotes a member m, which is declared in a class C. m may be a field or a method. In the latter case, m is assumed to include the whole method signature. Only members actually declared in class C can be denoted with $C.m$, members inherited from superclasses of C must be identified with the appropriate superclass name. $static(C.m)$ denotes a boolean expression which indicates whether $C.m$ has been declared using the keyword `static`. Further $p, q, \ldots$ indicate variables in $\mathcal{P}$ and $x, y, \ldots$ expressions. $s, t, \ldots$ designate object creation sites (details are explained in a moment). In the following definitions, $type(v)$ denotes the static type of a variable, expression or object creation site v. The relational operator $\leq$ is used for checking of subtypes: $c \leq c'$ is true iff an expression of type c can be assigned to a reference of type c'.

The *objects* $\mathcal{O}$ are the variables and objects of the program. Variables whose type is not a reference, but a base-type can be ignored, since these variables cannot be used to access members. Since in Java objects are created by the `new` operator, they do not have variable names like a stack allocated objects in C++. The object creation site (e.g. a source code position) serves as an identifier for objects. Variables are identified by their name and those names must be unique. If the same variable name is used multiple times within a program (e.g. in different methods or in different blocks in the same methods), they must be made distinguishable. Implicit `this`-pointers are also included and they are given the variable name `<this>`[1].

[1] This was inspired by the names chosen for not explicitly named methods in Java bytecode: `<init>` and `<clinit>`

Definition 1 The sets of pointers and objects for a program $\mathcal{P}$ are defined as follows:

$$\begin{aligned} \mathit{Pointers}(\mathcal{P}) &:= \{v | v \text{ is a variable in } \mathcal{P}, \mathit{type}(v) = C, C \text{ is a class in } \mathcal{P}\} \\ \mathit{Objects}(\mathcal{P}) &:= \{s | \texttt{new C} \text{ in } \mathcal{P} \text{ at } s, \mathit{type}(s) = C, C \text{ is a class in } \mathcal{P}\} \end{aligned}$$

Pointers and objects are the objects for the concept analysis:

$$\begin{aligned} \mathit{Pointers} &\subseteq \mathcal{O} \\ \mathit{Objects} &\subseteq \mathcal{O} \end{aligned}$$

The *attributes* $\mathcal{A}$ are class members. For members *definitions* and *declarations* are distinguished. A declaration of a member only contains a prototype or an abstract declaration, while the definition also contains the actual implementation. This was first suggested by Tip and Sweeney [49, 52], and enables a more precise handling of virtual method calls, as information about the actual implementation used at run-time does not become visible at the call-site, because for the invocation only a declaration is needed. If no dynamic binding is used (for constructors, `private` methods or data-members) this distinction is not required and using only a definition is sufficient[2]. On the other hand, for an `abstract` method, only a declaration is needed.

Definition 2 Member declarations and definitions for a program $\mathcal{P}$ are defined as follows (constructors are not considered methods):

$$\begin{aligned} \mathit{MemberDcls}(\mathcal{P}) := \{&\mathit{dcl}(C.m) | m \text{ is a non private method in } C, \\ &C \text{ is a class in } \mathcal{P}\} \\ \mathit{MemberDefs}(\mathcal{P}) := \{&\mathit{def}(C.m) | m \text{ is not an abstract method in } C, \\ &C \text{ is a class in } \mathcal{P}\} \end{aligned}$$

Member declarations and definitions are attributes for the concept analysis:

$$\begin{aligned} \mathit{MemberDcls}(\mathcal{P}) &\subseteq \mathcal{A} \\ \mathit{MemberDefs}(\mathcal{P}) &\subseteq \mathcal{A} \end{aligned}$$

[2]The original Snelting/Tip created declarations for data-members. This was changed to be more consistent: declarations are used for access using dynamic binding, definitions are used for direct access. As the rules for declarations and definitions are almost symmetrical, this does not result in any differences besides the notation

2.2.2 Constructing the Table

Auxiliary Definitions

In order to get all possible targets for a dynamic method call $p.f()$, for each pointer $p \in \mathit{Pointers}$, an approximation of all objects $o \in \mathit{Objects}$ that *p may point to* is needed. There is a number of existing algorithms that are usable. The specific requirements will be discussed in chapter 4. The following definition represents the result required from any used algorithm:

Definition 3 The points-to information for a program $\mathcal{P}$ is defined as follows:

$$\mathit{PointsTo}(\mathcal{P}) := \{(p, o) | p \in \mathit{Pointers}(\mathcal{P}), o \in \mathit{Objects}(P), p \text{ may point to } o\}$$

Two simple algorithms shall be given as an example:

The first one simply assumes a pointer p may point to all objects of a program that have a matching type:

$$\mathit{PointsTo}(\mathcal{P}) := \{(p, o) | p \in \mathit{Pointers}(\mathcal{P}), o \in \mathit{Objects}(\mathcal{P}), \mathit{type}(o) \leq \mathit{type}(p)\}$$

Alternately, each object can be propagated through all assignments (for a definition of assignments see below):

$$\begin{aligned}\mathit{PointsTo}(\mathcal{P}) := \{(p, o) | p \in \mathit{Pointers}(\mathcal{P}), o \in \mathit{Objects}(\mathcal{P}),\\ \exists p_1, p_2, \ldots, p_n : p = p_1 \wedge p_n = o \wedge\\ \forall_{i=1}^{n-1}(p_i, p_{i+1}) \in \mathit{Assignments}(\mathcal{P})\}\end{aligned}$$

This is basically identical to Andersen's points-to analysis[3].

Table Entries for Member Access Operations

The table T has pointers and objects of the program $\mathcal{P}$ as rows and declaration and definition attributes for the class members of $\mathcal{P}$ as columns. Informally, a table entry $(p, \mathit{dcl}(C.m))$ is created, iff a declaration for m must be contained in p's type and an entry $(p, \mathit{def}(C.m))$, iff the definition of $C.m$ must be contained in p's type. These requirements are created by access to the members in the program. Static or dynamic access to members is distinguished. For dynamic access, the implementation used is determined with the type of the object at run-time, for static access the compile time type is used. Static access applies to data members, but also in calls to `private` methods and constructors. It is also applied, if a method is directly called on a object without using a reference[3].

[3] In Java this is only possible by writing something like `(new A()).f()`

Definition 4 The set of member access operators for a program $\mathcal{P}$ is defined as follows:

$$\begin{aligned}
&\mathit{StaticMemberAccesses}(\mathcal{P}) := \\
&\quad \{(m,p) | p.m \text{ occurs in } \mathcal{P}, p \in \text{Pointers}(\mathcal{P}), \\
&\qquad\qquad \nexists C : \mathit{dcl}(C.m) \in \mathit{MemberDcls}(\mathcal{P}) \wedge \mathit{type}(p) \leq C\} \cup \\
&\quad \{(m,s) | s.m \text{ occurs in } \mathcal{P}, s \in \text{Objects}(\mathcal{P}\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{DynamicMemberAccesses}(\mathcal{P}) := \\
&\quad \{(m,p) | p.m \text{ occurs in } \mathcal{P}, p \in \text{Pointers}(\mathcal{P}), \\
&\qquad\qquad \exists C : \mathit{dcl}(C.m) \in \mathit{MemberDcls}(\mathcal{P}) \wedge \mathit{type}(p) \leq C\}
\end{aligned}$$

For both kind of accesses, the actual member selected is determined by the static lookup. Static lookup traverses the class hierarchy from a starting class C in search of a declaration or definition of a member m. In Java, this is a two step process even for static members: the first step is done at compile time, the second step at run-time. If C is a class, the compiler checks for the existence of m in C and then puts a reference to $C.m$ into the code, regardless if m was declared in C itself or in a superclass or superinterface of C. If C is in an interface, any one declaration of m from any superinterface is chosen (e.g. if I and J are interfaces declaring f and K extends both I and J, lookup for f in K will result in $K.f$). In other words, the compiler only searches for a matching declaration. This lookup function will be referred to as *LookupDeclaration*. The definitions are then searched at run-time by the JVM. For static accesses lookup is started at the type the compiler has written into the bytecode, for dynamic accesses lookup is started at the type of the object used for access. As the JVM only searches for definitions, the ambiguity for multiple-inheritance interfaces is irrelevant. This lookup function will be denoted *LookupDefinition*. Details of the lookup functions can be found in [17] and [24].

Using these functions, table entries can be generated from the access sets:

Definition 5 The table entries for a program $\mathcal{P}$ due to member access operations are defined as follows:

$$\frac{(m,x) \in \mathit{StaticMemberAccess}(\mathcal{P})}{X := \mathit{LookupDefinition}(\mathit{type}(x), m), (x, \mathit{def}(X.m)) \in T}$$

$$\frac{(m,x) \in \mathit{DynamicMemberAccess}(\mathcal{P})}{X := \mathit{LookupDeclaration}(\mathit{type}(x), m), (x, \mathit{dcl}(X.m)) \in T}$$

$$\frac{(m,p) \in \mathit{DynamicMemberAccess}(\mathcal{P})}{\bigwedge_{o}^{(p,o)\in\text{PointsTo}(\mathcal{P})} X := \mathit{LookupDefinition}(\mathit{type}(o), m), (o, \mathit{def}(X.m)) \in T}$$

Table Entries for `this`-Pointers

A method $C.m$ is represented twice in the table. It has a column for a definition attribute and a row for the `this`-pointer. The first collects which other pointers and objects actually access $C.m$, while the latter one collects the class members accessed in the body of $C.m$ via the `this`-pointer. However, they do not have to appear at the same lattice element[4]. [43] shows that $\mu(\mathit{def}(C.m)) \leq \gamma(C.m.\texttt{<this>})$ must always hold. This means, the `this`-pointer may have a weaker type than the class its implementation is contained in. But the type of the `this`-pointer will be determined by the implementing class, so it is only consistent to ensure also $\gamma(C.m.\texttt{<this>}) \leq \mu(\mathit{def}(C.m))$ and force the `this`-pointer to appear at the same lattice element as the definition.

Definition 6 The following entries are added to the table for a program $\mathcal{P}$:

$$\frac{\mathit{def}(C.m) \in \mathit{MemberDefs}(\mathcal{P})}{(C.m.\texttt{<this>}, \mathit{def}(C.m)) \in T}$$

Implications for Assignments

The program $\mathcal{P}$ may contain assignments in the form $x = y$. If x and y are references or objects, $\mathit{type}(y) \leq \mathit{type}(x)$ must be valid. Of course, the assignment must still be valid for the new hierarchy, so $\gamma(\mathit{type}(y)) \leq \gamma(\mathit{type}(x))$ is required. As shown earlier, this property can be guaranteed by an implication $y \rightarrow x$. Informally, this implication can be described as "everything accessed by x must also be available in y".

A program contains assignments not only in explicit form, but also for implicit one like e.g. passing parameters to methods. For each method call, assignments between the formal and actual parameters of the method must be added. If the call is a dynamic call $p.m$, then the points-to sets can be used to resolve all possible targets for this function call:

$$\{C.m | (p,o) \in \mathit{PointsTo}(\mathcal{P}) \wedge \mathit{LookupDefinition}(\mathit{type}(o), m) = C\}$$

Additional assignments must also be made for `this`-pointers and return values (if their type is a reference).

[4] And it is the usual case as long as the method does not contain a static call to itself.

Definition 7 The set of assignments for a program $\mathcal{P}$ is defined as follows:

$$\begin{aligned} &\mathit{Assignments}(\mathcal{P}) := \\ &\quad \{(v,w) | v = w \text{ occurs in } \mathcal{P}, v \in \mathit{Pointers}(\mathcal{P}), \\ &\qquad w \in \mathit{Pointers}(\mathcal{P}) \cup \mathit{Objects}(\mathcal{P})\} \end{aligned}$$

From these assignments implications can be added to the table:

Definition 8 For a program $\mathcal{P}$ the following implications must be encoded into its table due to the assignments in $\mathcal{P}$:

$$\frac{(x,y) \in \mathit{Assignments}(\mathcal{P})}{x \rightarrow y}$$

If the sets of assignments contains a cycle, all variables contained in this cycle will appear at the same lattice element and thus have the same new type.

Implications for Lookup Behavior

So far it is guaranteed that the new types for all pointers and objects will contain all members accessed. In order for the program to keep its original behavior it is also required, that all member accesses in the new program access the same member as they did in the original program. If the original hierarchy contains two different members with identical signature and one hides the other (e.g. a method that overrides a method in its superclass), it may happen that this *hiding* is not automatically preserved, and in the new hierarchy the class could access a different member than it originally did. To keep that from happening, the following property must be fulfilled:

$$\mathit{LookupDefinition}(\mathit{type}(o), m) = C \Leftrightarrow \mathit{LookupDefinition}(\gamma(o), m) = C$$

A similar rule for the declarations can be omitted, as all declarations have the same semantics and only the definition used is relevant.

In Java, a member that is declared in a class will always hide a member from a superclass. To ensure this property, it is only required to force the definition attribute used in the original hierarchy to be below all other matching definitions. This can be enforced by the following rules:

Definition 9 The following implications are incorporated into a table T for a program $\mathcal{P}$ to preserve member hiding:

$$\frac{(x, \mathit{def}(A.m)) \in T, (x, \mathit{def}(B.m)) \in T, \quad B \leq A, A \neq B}{\mathit{def}(B.m) \rightarrow \mathit{def}(A.m)}$$

$$\frac{(x, dcl(A.m)) \in T, (x, def(B.m)) \in T, \quad B \leq A}{def(B.m) \rightarrow dcl(A.m)}$$

An implication is only created, if a table row with access to two conflicting members actually exists. The created implications always go from sub- to superclass and thus never create a cycle.

The original algorithm from Snelting/Tip used a system with four different kind of implications (additionally $dcl \rightarrow def$ and $dcl \rightarrow dcl$). This is required because C++ allows direct assignments between objects, whereas in Java an object can never be on the left hand side of an assignment. The more complex system had its roots in earlier work [53]. Of course, the original system can also be used for Java, but may add artificial structure to the class hierarchy and was therefore simplified.

While the requirement for $def \rightarrow def$ is quite obvious as the lookup of member definitions must be preserved, the requirement for $def \rightarrow dcl$ implications is more subtle. If an object accesses a member declaration and a definition which are both also accessed by different objects, those attributes will not be placed at the same lattice element as the object. Figure 2.1 shows an example program and figure 2.2(a) the corresponding lattice (without the implication). In this example, object $B1$ accesses $def(B.f)$ as result of the dynamic method call to f and $dcl(B.f)$ because $B1$ is assigned to b, which gets an entry for the call to f. The lookup of f for $B1$ (and $B2$ too) is ambiguous, as neither the declaration hides the definition or vice versa. A $def \rightarrow dcl$ implication however will force a conflicting declaration above the definition. The lattice with such an implication can be seen in figure 2.2(b), where the conflicting lookup is removed.

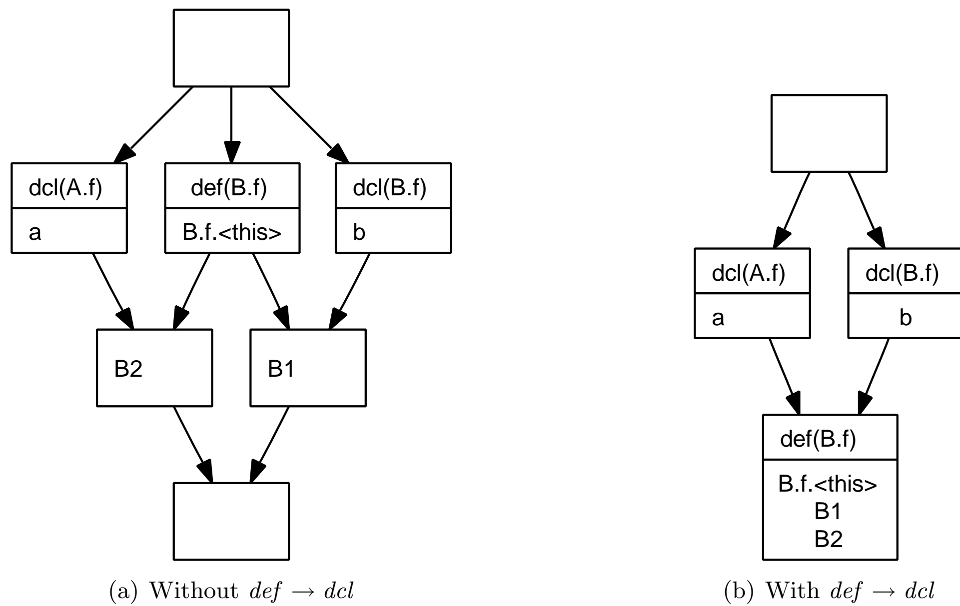
2.2.3 Constructing the Lattice

To create the final table, a fix-point iteration is required, as the assignments implications may create new hiding implications and vice versa[5]. For creation of the lattice, Ganther's algorithm will be used. Snelting and Tip state in [43] that the rows for `this`-pointers can be omitted from the table when the lattice is constructed without consequences because the row is redundant. Unfortunately, this proofed to be untrue. A row is only redundant, if another row with the same entries exists. Otherwise a row will always change the structure of the lattice. If the access pattern for a `this`-pointer is unique within the table, there will be a lattice element with the sole object label for the `this`-pointer. If the pointer is omitted during lattice construction, this element may vanish. Figure 2.3 shows a sample program, figure 2.4(a) the lattice with `this`-pointers included and figure 2.4(b) the lattice with `this`-pointers omitted. The difference is quite obvious. In the latter

[5] In practice the fix-point is usually reached after two or three iterations

```
1   class A {
2      void f() { }
3   }
4   class B extends A {
5      void f() { }
6   }
7
8   class Test {
9      public static void main(String args[]) {
10        B b=new B(); // B1
11        b.f();
12        A a=new B(); // B2
13        a.f();
14     }
15  }
```

Figure 2.1: $def \rightarrow dcl$ example program

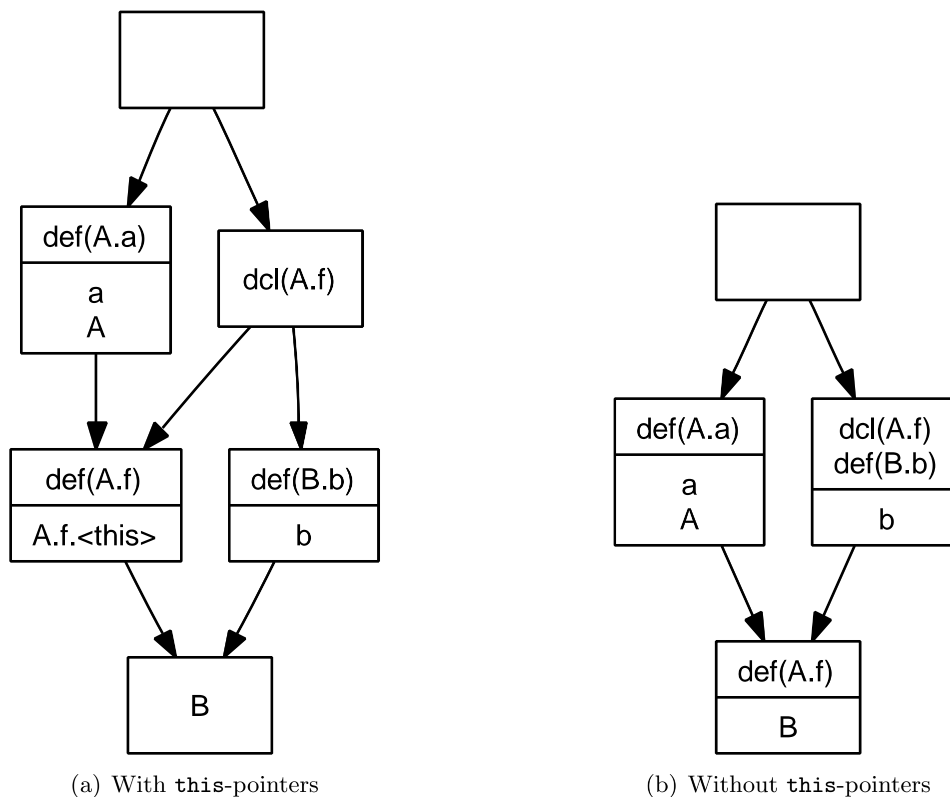


(a) Without $def \rightarrow dcl$ (b) With $def \rightarrow dcl$

Figure 2.2: Lattices for the $def \rightarrow dcl$ example program (2.1)

```
1   class A {
2      int a;
3      void f() { a=5; }
4   }
5   class B extends A {
6      int b;
7   }
8   class Test {
9      public static void main(String args[]) {
10        A a=new A(); // A1
11        a.a=27;
12        B b1=new B(); // B1
13        b.f();
14        b.b=61;
15     }
16  }
```

Figure 2.3: Omission of `this`-pointer example program



(a) With `this`-pointers

(b) Without `this`-pointers

Figure 2.4: Lattices for the omission of `this`-pointer example program (2.3)

variant, the new type for the `this`-pointer of $A.f$ would include the member $B.b$, which was in a subclass of A before, making the type of `this`-pointer actually larger than in the original program! To prevent that from happening, `this`-pointers must be left in the table for lattice construction. If they are omitted, the transformation of the program to the new class hierarchy may result in an type-incorrect program, because member entries may be missing for `this`-pointers.

In addition to that, the proof given by Snelting/Tip in [43] for the fact, that `this`-pointer and *def* attribute always appear at the same lattice element contains an omission. They assume that for a table entry $(x, def(C :: f)) \in T$ a method call `x.f()` must exist. For C++ this is wrong, as the table entry can also be created from a method call `p->f()` and $(p, x) \in PointsTo(\mathcal{P})$. However, it does not make a difference in practice, as in this case a table entry $(p, def(C :: f))$ and an assignment (x, p) will be created and $(x, def(C :: f)) \in T$ will be valid after the table iteration.

Snelting/Tip also showed that pointers in general can be removed from the table before creating the lattice. However, the effect shown with the `this`-pointer applies to omission of every other pointer too. There will always be lattice elements which provide a suitable replacement for the pointer, but in some cases all of these elements may be *larger* than required and there is no way to automatically determine which elements suits best. Choosing one of them randomly introduces exactly the kind of redundancies to the class hierarchy that Snelting/Tip was meant to remove.

Whether pointers should be omitted or not, depends on the purpose of the analysis. If it is used to study object behavior and to get some ideas for restructuring, omitting pointers is fine. For an automated transformation of code, exactly those pointers, which have an explicit type in the code must be added to the analysis. For transforming Java bytecode pointers for parameters are required in order to build new types for method signatures, but pointers for local variables in methods can be omitted, as the JDK will infer these types anyway. Only for the transformation of Java source code are all pointers required, as usually a pointer has a declared type somewhere in the source code and those type names have to be adjusted for a source code transformation.

Chapter 3

Extensions to Snelting/Tip

In this chapter, various modifications and enhancements for the original Snelting/Tip analysis will be shown. Most of these are required in order to analyze the full Java language instead of the original C++ subset, but not all of them are specific for Java.

3.1 Static Methods and Fields

The original analysis did not handle static methods and fields. As access to them is done without an object reference, their containing class is of lower significance than for non-static members, but still relevant, even if only for the decision as to whether a static member is dead or alive.

In the spirit of the analysis the same attributes as for non-static members are generated for static members:

Definition 10 Additional member definitions for the static members of a program $\mathcal{P}$ are defined as follows:

$$\{def(C.m) | m \text{ is a static field or method in class } C\} \subseteq MemberDefs(\mathcal{P})$$

As access to these members does not use an object reference, the corresponding table columns will not get entries automatically, so an additional table row is also needed.

Definition 11 Additional pointers for access for to static members of a program $\mathcal{P}$ is defined as follows:

$$\{static(C.m) | m \text{ is a static field or method in class } C\} \subseteq ClassPtrVars(\mathcal{P})$$

Table entries are generated for each read or write access to a static member:

Definition 12 Additional table entries for static members of a program $\mathcal{P}$ are defined as follows:

$$\frac{C.m \text{ occurs in } P,\ m \text{ is a static class member of } C \text{ in } P}{(static(C.m), def(C.m)) \in T}$$

This will basically create an individual class for each used static member. They can later be manually merged together with the classes containing non-static members.

The granularity of this approach may be changed by choosing a different granularity for the accessing variables. E.g. only a single object $static(C)$ can be used for all members of the same original class. In this case, different definitions for the creation of table rows and entries must be used:

Definition 13 Alternate additional pointers for access for to static members of a program $\mathcal{P}$ is defined as follows:

$$\{static(C.m)|m \text{ is a static field or method in class } C\} \subseteq ClassPtrVars(\mathcal{P})$$

Definition 14 Alternate table entries for static members of a program $\mathcal{P}$ are defined as follows:

$$\frac{C.m \text{ occurs in } P,\ m \text{ is a static class member of } C \text{ in } P}{(static(C), def(C.m)) \in T}$$

3.2 Class Initializers

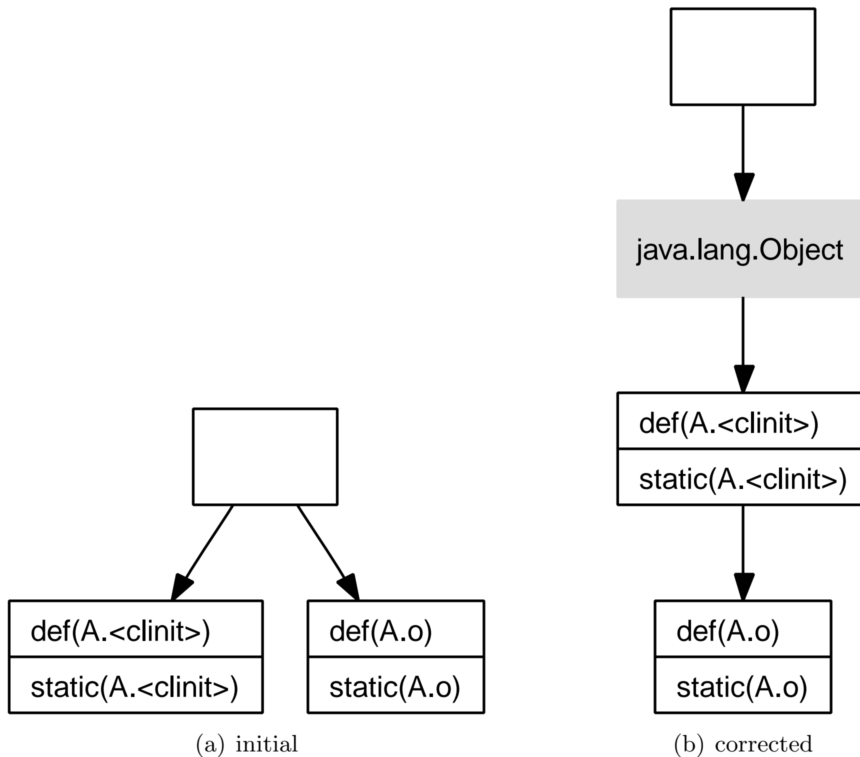
Java provides a mechanism to execute a piece of code if a class is loaded into the virtual machine. This code, together with all code for the initialization of static class members, is written into a special method, called the class initializer. The bytecode name of this method is `<clinit>` and this name will be used within class hierarchies.

The special thing about class initializers is that they are implicitly called if a class is loaded, so the analysis must deal with them in a special way or required code may not be executed in a new class hierarchy. Figure 3.1 shows an example of code that will break if class initializers are not handled appropriately. In the resulting class hierarchy (figure 3.2(a)) the class initializer and the static field are placed in different classes, but the main program will only reference the class containing the field, and thus the field never gets initialized.

To preserve the semantics of the original program, it must be ensured, that for any statement which might result in a call to class initializer, the same class initializer will be called in the new hierarchy. A class initializer for a class t is called if an instance of t is created, a static member of t is accessed or the class initializer for a subclass of t is called. Enforcing calls to the class initializer can be achieved by the following rules:

```
1   class A {
2      static Object o;
3      static {
4         o=new Object();
5      }
6   }
7
8   class Test {
9      public static void main(String args[]) {
10        A.o.hashCode();
11     }
12  }
```

Figure 3.1: Example program for class initializers



(a) initial (b) corrected

Figure 3.2: Analysis results for figure 3.1

```
1   class A {
2      static int a=B.b++;
3   }
4
5   class B {
6      static int b=A.a++;
7   }
8
9   class TestAB {
10     public static void main(String[] args) {
11        System.out.println(A.a);
12        System.out.println(B.b);
13     }
14  }
15
16  class TestBA {
17     public static void main(String[] args) {
18        System.out.println(B.b);
19        System.out.println(A.a);
20     }
21  }
```

Figure 3.3: Cyclic initialization of classes

Definition 15 The following constraints are generated for the class initializers of a program $\mathcal{P}$:

$$\frac{\texttt{new C} \text{ occurs in } P}{\mathit{def}(C.\texttt{<init>}) \rightarrow \mathit{def}(C.\texttt{<clinit>})}$$

$$\frac{C.m \text{ occurs in } P,\ m \text{ is a static class member of } C \text{ in } P}{\mathit{def}(C.m) \rightarrow \mathit{def}(C.\texttt{<clinit>})}$$

$$\frac{A \text{ is a direct base class of } B}{\mathit{def}(B.\texttt{<clinit>}) \rightarrow \mathit{def}(\texttt{A.<clinit>})}$$

For the given example program, the correct class hierarchy is shown in figure 3.2(b). In this hierarchy access to `A.o` results in a call to the class initializer of `A` (which in turn calls the class initializer of `java.lang.Object`, so `Object` is included in this hierarchy).

This will not guarantee that the calling of class initializers will happen in the same order as in the original program, because the analysis does not make assumptions about the order of class loading. There are cases where the order of loading the classes defines the semantics of the program. The program in figure 3.3 provides a two classes with a circular dependency

in their class initializers and two `main` methods. Both clients produce the same output, although the fields are printed in different order. Fortunately such effects can be considered obscure and are not present in casual Java programs.

3.3 Runtime Type Checks

Java provides a mechanism which enables a programmer to check at runtime, whether the type of an object is equal to or a subtype of a given type with the `instanceof` operator. The actual syntax is o `instanceof` T, where o is a reference and T the name of a type.

In order for new class hierarchy to be semantically equivalent to the original hierarchy, the analysis must create a new type T' as a replacement for T, which has the following properties:

$$p \texttt{ instanceof } T \in P \Leftrightarrow \bigwedge_{o}^{o \in PointsTo(p)} \mathit{type}(o) \leq T \Leftrightarrow \gamma(o) \leq T'$$

Every `instanceof` operator divides all objects into three equivalence classes:

- Objects not tested against that operator
- Objects tested with result `true`
- Objects tested with result `false`

In order to create these three classes within the concept lattice, two additional attributes must be used:

Definition 16 The following attributes and table entries are generated for the use of the `instanceof`-operation in a program $\mathcal{P}$:

$$\frac{p \texttt{ instanceof } T \in P \text{ at } pc \text{ in method } M}{\{\mathit{check}(\mathit{instanceof}, M, pc, \mathit{true}), \mathit{check}(\mathit{instanceof}, M, pc, \mathit{false})\} \subseteq \mathcal{A}}$$

$$\bigwedge_{o}^{o \in PointsTo(p)} \begin{cases} (o, \mathit{check}(\mathit{instanceof}, M, pc, \mathit{true})) \in T & \mathit{type}(o) \leq T \\ (o, \mathit{check}(\mathit{instanceof}, M, pc, \mathit{false})) \in T & \text{else} \end{cases}$$

Inclusion of the position M, pc results in an increased analysis precision, as for every `instanceof` an individual attribute is used and this an individual type is calculated.

In the final concept lattice $\mu(\mathit{check}(\mathit{instanceof}, M, pc, \mathit{true}))$ can be used as the type T' as it satisfies the requirement given above. For all objects $o \in$

$\mathit{PointsTo}(p)$, a table entry is generated iff $\mathit{type}(o) \leq T$, so the requirement can be replaced by

$$(o, \mathit{check}(\mathit{instanceof}, M, pc, \mathit{true})) \in T \Leftrightarrow \gamma(o) \leq \mu(\mathit{check}(\mathit{instanceof}, M, pc, \mathit{true}))$$

This relation between table entry and lattice nodes is exactly what lattice construction guarantees. If p `instanceof` T or $o \in \mathit{PointsTo}(p)$ is not true, no entry $\mathit{check}(\mathit{instanceof}, M, pc, \mathit{true})$ will be generated, so $\gamma(o) \not\leq T'$ is also guaranteed.

As no assumptions were made about the $\mathit{check}(\mathit{instanceof}, M, pc, \mathit{false})$ attribute, it is clear that this attributes is not required to build a semantically equivalent class hierarchy. However, it is extremely useful later during interactive modification of the class hierarchy. Without the additional attribute, it would be impossible to differ between objects that may be moved below T' (as they are not in the points-to set of p and will never been tested against that particular `instanceof`) and objects which may not (as moving them below T' would change the result of `instanceof` at run-time).

Figure 3.4 shows an example program and figure 3.5 the resulting class hierarchy. It is easy to see that the original classes A got vertically split into a variant tested with `instanceof` and a variant not tested. This looks redundant at first (if the *false* attribute is omitted, the original class hierarchy will be reproduced), but without the attribute it would be impossible to merge the class containing $B1$ and $C1$, as the information that $B1$ has never been checked by the `instanceof` operator is not available and it has to be conservatively assumed, that $B1$ has been checked and program semantics will be broken if $B1$ and $C1$ have the same new type.

The new attributes may appear at $\bot$ in the new hierarchy. This must be handled specially when it comes to a source code transformation, as $\bot$ is not a valid type that can be used in a program. There are two special cases: (1) $\mu(\mathit{check}(\mathit{instanceof}, M, pc, \mathit{true})) = \bot \wedge \mu(\mathit{check}(\mathit{instanceof}, M, pc, \mathit{false})) \neq \bot$. In this case no tested object is assignment compatible to T, which makes it possible to replace the whole `instanceof` operator by `false`. (2) $\mu(\mathit{check}(\mathit{instanceof}, M, pc, \mathit{false})) = \bot$. In this case all objects tested are assignment compatible, but as during program execution the value of `p` could be `null`, the operator cannot be replaced by `true`, but by `p!=null`.

The dual attribute approach is based on an earlier idea presented in [43]. There `instanceof` is transformed into a method call, which is resolved to a method returning either `true` or `false`. The *def*-attributes for these two method correspond to the two attributes used here. Direct use of the attribute makes it easier to handle every use of `instanceof` individually. For the method approach, a context sensitive analysis of the transformed `instanceof` operators is required.

```
1   class A {
2   }
3   class B extends A {
4   }
5   class C extends B {
6   }
7
8   class Test {
9      static A a1, a2;
10
11     public static void main(String args[]) {
12        a1=new A();
13        a1=new C();
14        a2=new A();
15        a2=new B();
16
17        if(a1 instanceof C) {
18        }
19     }
20  }
```

Figure 3.4: Example program for `instanceof` operator

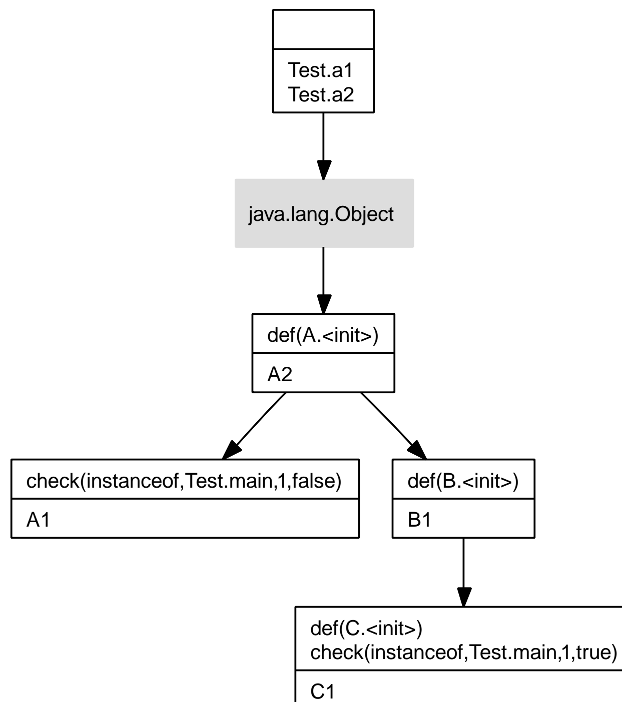


Figure 3.5: Class hierarchy for figure 3.4

3.4 Type-Casts

Type-casts in Java are similar to the `instanceof` operator. They differ in the handling of a `null` reference, which is irrelevant for this analysis, and their return value. While the `instanceof` operator returns a boolean value, a type-cast returns the reference in case of success or throws an exception if the object is not assignment compatible to the target type.

In terms of the original Snelting/Tip analysis a type-cast is a conditional assignment.

Definition 17 Type-casts in a program $\mathcal{P}$ create assignments as follows:

$$\frac{\texttt{q=(T)p} \text{ occurs in } \mathcal{P}}{\bigwedge_{o}^{o \in PointsTo(p)} \quad type(o) \leq T \Rightarrow (q, o) \in Assignments(\mathcal{P})}$$

Like for `instanceof` the calculation of a new type for T is done by creating two additional attributes:

Definition 18 The following attributes and table entries are generated for the use of type-casts in a program `P`:

$$\frac{\texttt{(T)p} \text{ occurs in } \mathcal{P} \text{ at } pc \text{ in method } M}{\begin{array}{c}\{(o, check(cast, M, pc, true), (o, check(cast, M, pc, false)\} \subseteq \mathcal{A} \\ \bigwedge_{o}^{o \in PointsTo(p)} \begin{cases} (o, check(cast, M, pc, true)) \in T & type(o) \leq T \\ (o, check(cast, M, pc, false)) \in T & \text{else} \end{cases}\end{array}}$$

The type $\mu(check(cast, M, pc, true))$ is then usable as target type for the cast (for the same reasons as the attributes for `instanceof` were in the previous section). The special cases however are different.

If $\mu(check(cast, M, pc, false)) = \bot$, the cast is always successful. Unfortunately this is not sufficient to remove that cast, as it does not guarantee $\gamma(p) \leq \gamma(q)$. Only if this additional condition is met can the cast be removed completely and replaced by a simple assignment.

If $\mu(check(cast, M, pc, true)) = \bot$, the cast is not successful for any tested object. As `null` may still pass the cast, the cast must be replaced by this fragment:

```
if(p!=0) throw new CastClassException();
q=null;
```

The type $\gamma(p)$ is unsuitable, as this type may be too general and objects not passing the original cast can still be assignment compatible to $\gamma(p)$.

In order for the new cast to be legal Java code, the type of the left hand side q must be assignment compatible to $\mu(check(cast, M, pc, true))$,

which means $\gamma(q) \leq \mu(check(cast, M, pc, true))$ must be true. This is always true, because $ext(\mu(check(cast, M, pc, true)))$[1] contains exactly those objects which are assigned to q, which places them below $\gamma(q)$ and enforces the assignment compatibility.

3.5 Exception Handling

Exception handling is the third variant of run-time type-checks. Here an exception object is checked against a list of exception handlers. Each handler as an assorted type and if the object has that or a subtype, execution is continued at the code for that handler. If the exception matches no handler, execution of the current method is terminated, and exception handling is continued at a previous stack frame. If the exception is not caught at all, the program is terminated.

Intra-procedurally, for each statement in the program, a list of exception handlers $(e_1, t_1, v_1), ..., (e_n, t_n, v_n)$ can be given, where e_i is the number of this handler within the exception handler table, t_i is the caught type and v_i the variable used to catch the exception. The additional information where in the program the control flow is resumed, is omitted here as it is not relevant for the analysis. To preserve the semantics for this operation, the following condition must be fulfilled (where t'_i is the type of t_i in the new hierarchy):

$$\bigvee_m \Big(\bigwedge_{i=1}^{m-1} type(o) \not\leq t_i\Big) \wedge (m = n \vee type(o) \leq t_m) \Leftrightarrow \\ \Big(\bigwedge_{i=0}^{m-1} \gamma(o) \not\leq t'_i\Big) \wedge (m = n \vee \gamma(o) \leq t'_m)$$

Informally spoken, the exception must be caught by same handler in the new hierarchy as it was caught by in the old hierarchy.

As handling of exceptions is basically identical to a multiple use of the `instanceof` operator, similar attributes are used:

Definition 19 The following attributes and table entries are generated for exception handling in a program `P`:

[1] $ext(c) = \mathcal{O}$ for a concept $c = (\mathcal{O}, \mathcal{A})$

$$\frac{\begin{array}{c}o \text{ is thrown as exception by an instruction}\\ \text{with handlers } (e_1, t_1, v_1), \ldots, (e_n, t_n, v_n) \text{ in a method } M \text{ of } \mathcal{P}\end{array}}{\begin{array}{c}\bigwedge_{i=1..n} \mathit{check}(\mathit{exception}, M, e_i, \mathit{true}) \in \mathcal{A}\\ \bigwedge_{i=1..n} \mathit{check}(\mathit{exception}, M, e_i, \mathit{false}) \in \mathcal{A}\\ \bigwedge_{i=1..n}(\bigwedge_{j=1..i} \mathit{type}(o) \not\leq t_j) \Rightarrow (o, \mathit{check}(\mathit{exception}, M, e_i, \mathit{false}) \in T)\\ \bigwedge_{i=1..n}(\bigwedge_{j=1..i-1} \mathit{type}(o) \not\leq t_j) \wedge \mathit{type}(o) \leq t_i \Rightarrow\\ \hfill (o, \mathit{check}(\mathit{exception}, M, e_i, \mathit{true})) \in T\\ \bigwedge_{i=1..n}(\bigwedge_{j=1..i-1} \mathit{type}(o) \not\leq t_j) \wedge \mathit{type}(o) \leq t_i \Rightarrow\\ \hfill (v_i, o) \in \mathit{Assignments}(\mathcal{P})\\ (\bigwedge_{j=1..n} \mathit{type}(o) \not\leq t_j) \Rightarrow (M.\texttt{<exceptions>}, o) \in \mathit{Assignments}(\mathcal{P})\end{array}}$$

The definition is made up of multiple parts. First, attributes for exception handlers are created. As a new type for every exception handler is needed, the attributes include the number of the table entry. While M, pc served as additional information to increase the precision for `instance` and type-casts, M, e_i entries here are required to distinguish different exception handlers within the same method. Secondly, assume the exception is caught by handler number m. Then for $i = 1..m-1$ table entries for the *false* attribute and for m a table entry for the *true* attribute are created. No entries are created for handlers after m. Finally, if the exception is not caught by any handler, it is stored in a special variable to indicate, that an invocation of M may result in o being thrown as an exception. These variables can be used to determine, which exceptions a method may throw.

An exception handler e_i, t_i, v_i will usually occur in the list of handlers for different instructions, but for each e_i, t_i and v_i will be the same by definition of the exception handler table. This is a requirement for this approach to be usable. Otherwise table entries $(o, \mathit{check}(\mathit{exception}, M, e_i, \mathit{true}))$ as well as $(o, \mathit{check}(\mathit{exception}, M, e_i, \mathit{false}))$ could be required, as different t_i in order to check if e_i will catch o.

Again, the $false$ attributes are not required, although they may look more important here, but are even more useful for interactive refactoring. There is only one special case, if $\mathit{check}(\mathit{exception}, M, e_i, \mathit{true}) = \bot$. Then no exception is caught by this handler and its code can be removed during a source-code transformation.

3.6 Equality of Signatures for Overridden Methods

If the implementation of a methods is to be overridden in a subclass, languages like Java and C++ require the new implementation to have the same signature as the overridden method[2].

[2] C++ allows the overriding method to have a stronger return type. From JDK 1.5 on, Java allows this too, but KABA is based on 1.3, so this new feature remains unused.

```
1   class A {
2      void f() {}
3      void g() {}
4      void h() {}
5   }
6
7   class B {
8      void f(A a) { a.f(); }
9   }
10  class C extends B {
11     void f(A a) { }
12  }
13  class D extends C {
14     void f(A a) { a.f(); a.g(); }
15  }
16  class E extends B {
17     void f(A a) { a.h(); }
18  }
19
20  class Test {
21     static B b;
22
23     public static void main(String args[]) {
24        b=new B();
25        b=new C();
26        b=new D();
27        b=new E();
28
29        b.f(new A());
30     }
31  }
```

Figure 3.6: Example program illustrating signature problem

The original analysis takes no special caution to ensure this requirement will be fulfilled by the resulting class hierarchy. A small example program (figure 3.6) and the resulting class hierarchy (figure 3.7) will illustrate this problem.

The `param0` pointers refer to the first formal parameter of the method `f` in the corresponding class. All of these pointers access different members of `A` and so they all appear at different classes in the class hierarchy. If one tries to rewrite the original source code to the new hierarchy using these types, it results in an uncompilable program, because the interfaces seem

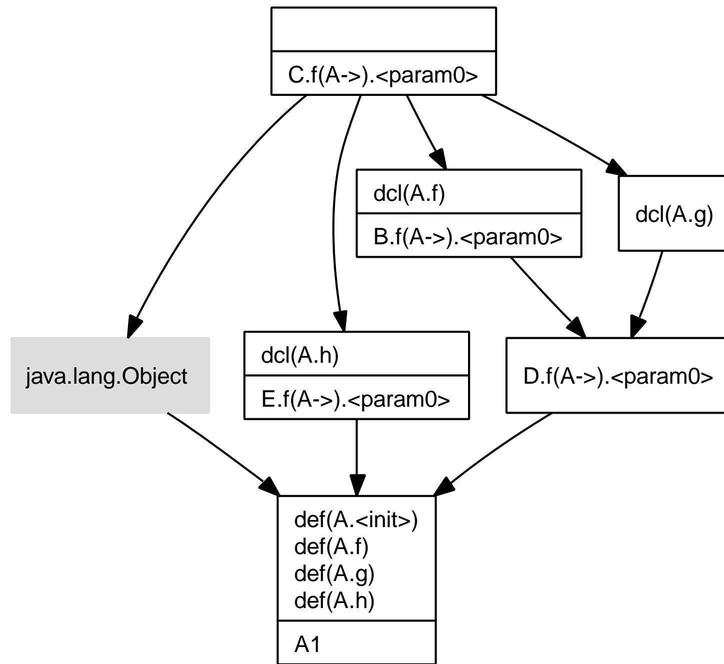


Figure 3.7: Original analysis of 3.6

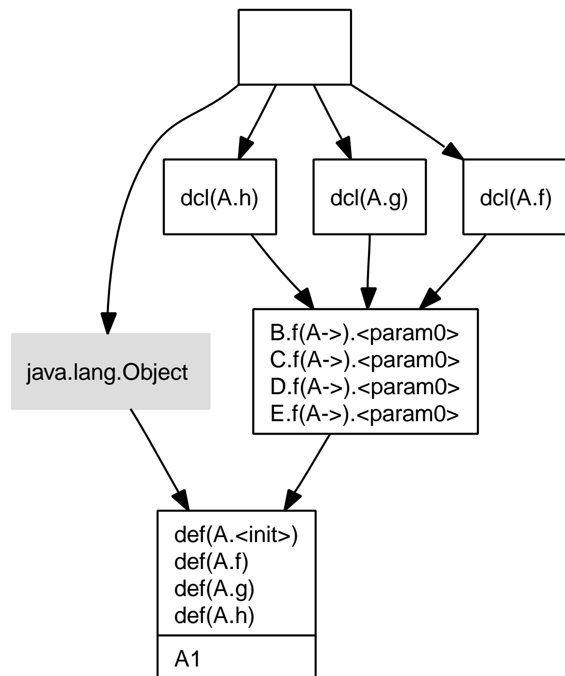


Figure 3.8: Analysis of 3.6 with enforcement of equal signatures

never to be implemented, as the parameter of the implementation will have a different type than the declaration in the interface.

A possibility to avoid this effect is to enforce that the same parameter for different implementations of a methods will have the same type in the new class hierarchy. This can be easily achieved by assigning them to each other (which is always possible, as they have the same type), creating an union-type of all the individual parameter types. Of course this is only required, if overriding methods are used together somewhere in the program. If they are not, their signatures do not need to be equal as the fact that one of them overrides the other is never used in the program. The same criteria is used for the creation of hiding implications, so these implications can be used to trigger the merging of the signatures.

$$\frac{def(B.m) \rightarrow def(A.m) \vee def(B.m) \rightarrow dcl(A.m)}{\begin{array}{l} merge(B.m.return, A.m.return) \\ \bigwedge_{i=0}^{n} merge(B.m.param_i, A.m.param_i) \end{array}}$$

$$\frac{merge(a,b)}{\begin{array}{c}(a,b) \in Assignments(\mathcal{P}) \\ (b,a) \in Assignments(\mathcal{P})\end{array}}$$

The result of these rules on the example program can seen in figure 3.8.

An alternative approach to this problem [47] is only slightly different. Instead of making two signatures completely equal, a subtype relation for the two parameters was enforced (using just a single assignment instead of two), resulting in a common subtype for all parameter types. The results of the example can be seen in 3.9. This common subtype (the type at which the parameter pointer for `B.f` is located) can be used as parameter type for all method signatures, because changing the type of a pointer to a subtype will not affect the program semantics.

But showing certain types for pointers within the class hierarchy and afterward using different types for rewriting the Java code is not a good idea, as it undoubtedly confuses the user of the analysis. For the given example, the first approach seems very invasive compared to the second. But in many cases the level of detail in the second example will be too high anyway and merging these pointers seems a logical step towards a smaller hierarchy.

3.7 Multiple Implementations in the Same Class

Figure 3.10 shows two classes where an overwritten method makes a call to the original implementation in the superclass. If only the derived class is instantiated, this will make both implementations appear at the same node

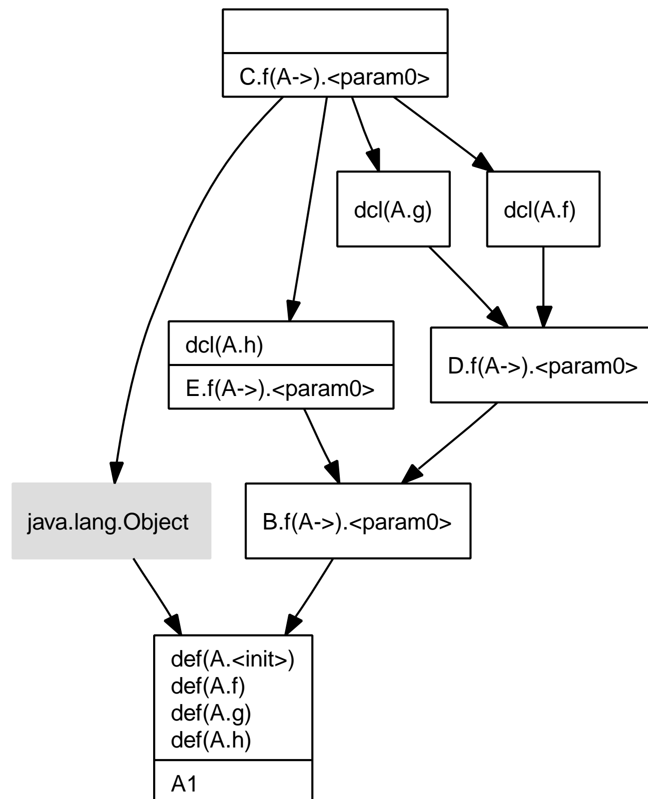
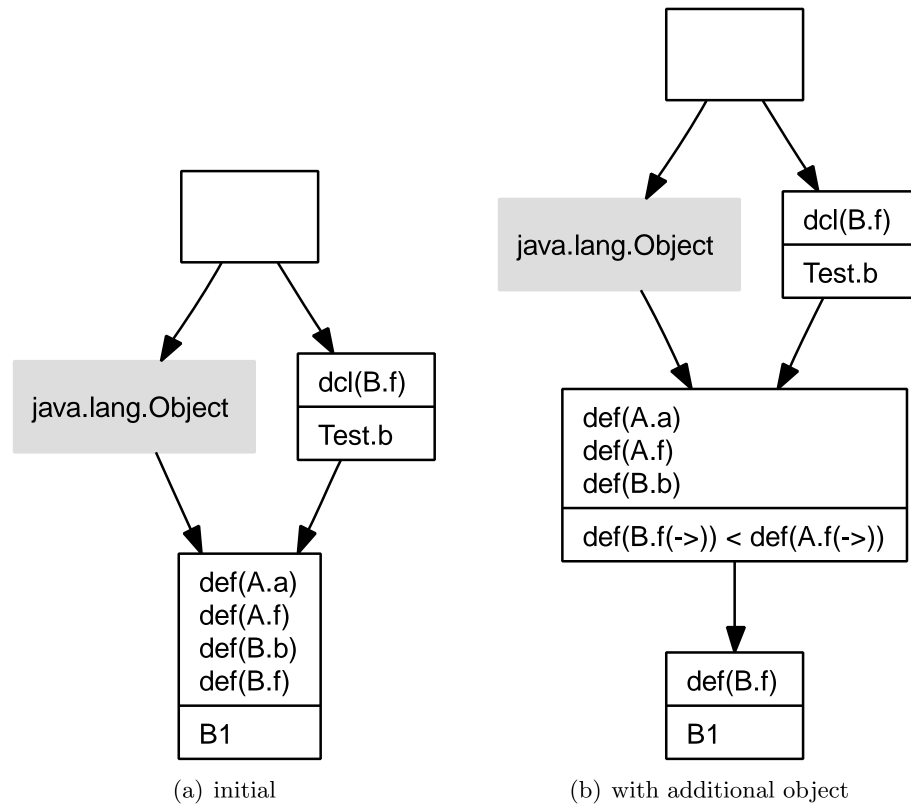


Figure 3.9: Analysis of 3.6 with common subtype for signatures

```
1   class A {
2      int a;
3      void f() {
4         a=0;
5      }
6   }
7
8   class B extends A {
9      int b;
10     void f() {
11        b=0;
12        super.f();
13     }
14  }
15
16  class Test {
17     static B b;
18
19     public static void main(String args[]) {
20        b=new B();
21        b.f();
22     }
23  }
```

Figure 3.10: Example for multiple implementations in one class



(a) initial (b) with additional object

Figure 3.11: Analysis result for figure 3.10

in the concept lattice. Figure 3.11(a) shows such a class hierarchy. It cannot be directly translated back into source code.

There are two possible solutions for this problem. (1) The method $A.f$ can be renamed. As there is no object in the program which accesses only $A.f$ but not $B.f$, no dynamically bound method call to f in the program may result in the implementation in A being executed, this is a safe operation. It might however be problematic if this is tried with constructors, as "renaming" a constructor would make it a common method and that is likely to affect its semantics. If the program contains a method call $x.f()$ that can be resolved to either $A.f$ or $B.f$, renaming one of the two methods becomes impossible. (2) As a better alternative, the table can be modified in order to make both implementations appear at different nodes. This seems more suitable for the Snelting/Tip analysis which was not meant to modify programs at statement level and works for constructors in the same way as it does for methods and has no other limitations.

A table row which separates $def(A.f)$ from $def(B.f)$ can be created as follows:

Definition 20 The following objects and table entries are added in order to prevent multiple methods with identical signatures to appear at the same lattice element for a program $\mathcal{P}$:

$$\frac{\begin{array}{c} A \text{ is a transitive base class of } B \\ \bigwedge_{o \in Objects} (o, def(A.m)) \in T \Leftrightarrow (o, def(B.m)) \in T \end{array}}{\begin{array}{c} r := def(B.f) < def(A.f) \\ r \in \mathcal{O} \\ \bigwedge_{a}^{a \in \sigma(\{o | (o, def(A.f)) \in T\}) \setminus \{def(B.f)\}} (r, a) \in T \end{array}}$$

The resulting class hierarchy can be seen in figure 3.11(b). Other attributes which are accessed by the same set of objects as the conflicting method definitions will be placed with the superclass method. This is required as at this point it is not possible to determine which of these members are used by which method, so they must be available for both implementations.

3.8 Library Classes

The original Snelting/Tip analysis always restructures the whole program code analyzed. But most of the time it is more useful, to restructure just a few classes and leave other classes the way they are. Maybe these classes do not require refactoring or modifying them is beyond control of the user of the analysis. For the latter case, the classes that come with the JDK are a perfect example. These classes are used in every Java program, but a usual requirement for the restructured program would be to run on a standard JDK with standard library classes and not to rely on a restructured version of the Java API.

Which classes are subject to refactoring and which classes are meant to be unchanged by the library, must be specified by the user. In the following section, a function $Library(c)$ is assumed, which evaluates to true if c is a library class and false otherwise. Library classes are expected not to *know* about the non-library classes. This means, they may not use them as superclasses, create instances, access fields or call methods. E.g. tagging all classes from the JDK as library and all classes with self-written program code as non-library fulfills this condition[3]. Classes and code that are not part of the library are referred to as *user* classes and code.

If p is a pointer or an object, $Library(p)$ must return true iff the type of p cannot be changed by the user. This is automatically true for all pointers or objects that are declared inside library classes, but also for the types of

[3]Reflection is a exception, but it is handled differently anyway

```
1   import java.util.*;
2
3   class A {
4      public int hashCode() { return 42; }
5      public boolean equals(Object o) { return true; }
6   }
7
8   class Test {
9      public static void main(String args[]) {
10         Set<A> set=new HashSet<A>();
11
12         set.add(new A());
13         set.add(new A());
14
15         System.err.println(set.size());
16     }
17  }
```

Figure 3.12: Example program for library classes calling user code

exceptions thrown by the JVM (e.g. a `java.lang.NullPointerException` when a member is accessed through a `null`-reference).

Besides leaving library classes unmodified, there is a second motivation for handling them differently. A usual Java program uses a lot of library classes and analyzing them all makes the analysis very expensive. If they do not need to be refactored, it seems worthwhile to check if the overhead created for them may be reduced a bit.

3.8.1 Reducing Table Entries for Library Classes

The most simple approach would be to simply ignore library classes or at least code from library classes. Not surprisingly, doing so results in a broken analysis. Figure 3.12 shows a small problem where the methods `hashCode` and `equals` are overwritten to ensure that only one instance of the class can be stored in a `HashMap`. The calls to these methods appear only in library code, not inside the program itself. If the library code is not analyzed, the two methods will be marked as dead, and thus break the restructured program.

If the code from library classes must be analyzed, it may be possible to reduce its effects on the analysis and exclude it from the table. Including code into the analysis usually results in an increased table size and maybe this can be avoided by ignoring pointers or objects of library functions.

Unfortunately ignoring objects proved impossible too, because they may

```
1   class LibraryA {
2      static LibraryA create() {
3         return new LibraryA();
4      }
5   }
6   class A extends LibraryA {
7   }
8   class Test {
9     static Object o;
10    public static void main(String[] args) {
11      o=LibraryA.create();
12      o=new A();
13
14      if(o instanceof A) {
15      }
16    }
17  }
```

Figure 3.13: Example program for library objects influencing user code

influence type-checks. Figure 3.13 shows a small program, where `LibraryA` represents a library class. If the instance of that class is ignored for the analysis, the new type for the `instanceof` check may be too general (as then the check is true for every object checked) and allow the `LibraryA`-Object to pass too, breaking the original program.

For pointers the situation is a little better. Certain pointers resulting from library code can be ignored like ordinary pointers can be. They must be part of the analysis and be present during table construction but can be eliminated before constructing the concept lattice. Library pointers may create additional type-constraints on other pointers (see below for details). Of course these constraints must be added to the table before pointers are eventually eliminated.

In addition to table rows, library code also creates additional columns for the declarations and definitions of analyzed members. Obviously they cannot be left out completely, but it seems possible to have only a single column, representing the entire class instead of individual columns for all members, as the class is meant to be unchanged anyway.

Enforcing all members to be at the same class can be done with an additional attribute per class (called the *class attribute* henceforth) and a set of constraints:

$$\frac{\mathit{Library}(C)}{\mathit{all}(C) \in \mathcal{A}}$$

```
1   class A {
2      public String toString() {
3         return "an A ("+super.toString()+")";
4      }
5   }
6
7   class Test {
8      public static void main(String args[]) {
9         A a1=new A();
10        A a2=new A();
11
12        a1.toString();
13     }
14  }
```

Figure 3.14: Example program for imprecision by collapsed library classes

$$\frac{\mathit{Library}(C), \mathit{def}(C.m) \in \mathcal{A}}{\begin{array}{c}\mathit{all}(C) \rightarrow \mathit{def}(C.m)\\ \mathit{def}(C.m) \rightarrow \mathit{all}(C)\end{array}}$$

$$\frac{\mathit{Library}(C), \mathit{dcl}(C.m) \in \mathcal{A}}{\begin{array}{c}\mathit{all}(C) \rightarrow \mathit{dcl}(C.m)\\ \mathit{dcl}(C.m) \rightarrow \mathit{all}(C)\end{array}}$$

These constraints force the rows for all declarations and definitions of a class to be identical, so they can be represented by just a single row.

Figure 3.14 shows an example program and figure 3.15 shows the resulting class hierarchy from an analysis using these additional constraints. The result looks as expected. `java.lang.Object` is represented as a single class, containing all its individual members and both `A` objects are in subclasses, and the `toString` method is only contained in the object that actually used it in the original program.

The problem with this approach only becomes visible later, if during interactive manipulation the user tries to merge the two classes for $A1$ and $A2$. In this case he will get an error message explaining, that this operation would change the lookup of `toString` from the implementation in `java.lang.Object` to the implementation in `A`, which is not allowed. Obviously this is nonsense, as both `A`-objects had used the implementation in `A`. However, this knowledge is no longer present in the class hierarchy (or the table) and so it is conservatively assumed, that $A2$ needs the implementation in `java.lang.Object`.

This effect is caused by forcing all attributes of a class to a single node, which results in a reduced level of precision of member accesses. Both

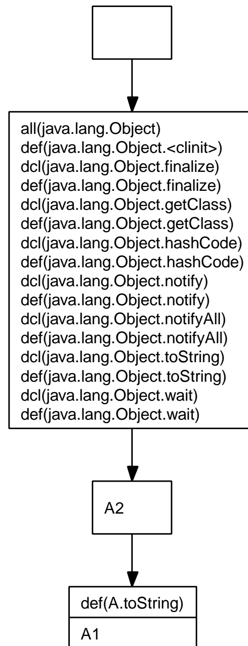


Figure 3.15: Class hierarchy for figure 3.14 with collapsed library classes

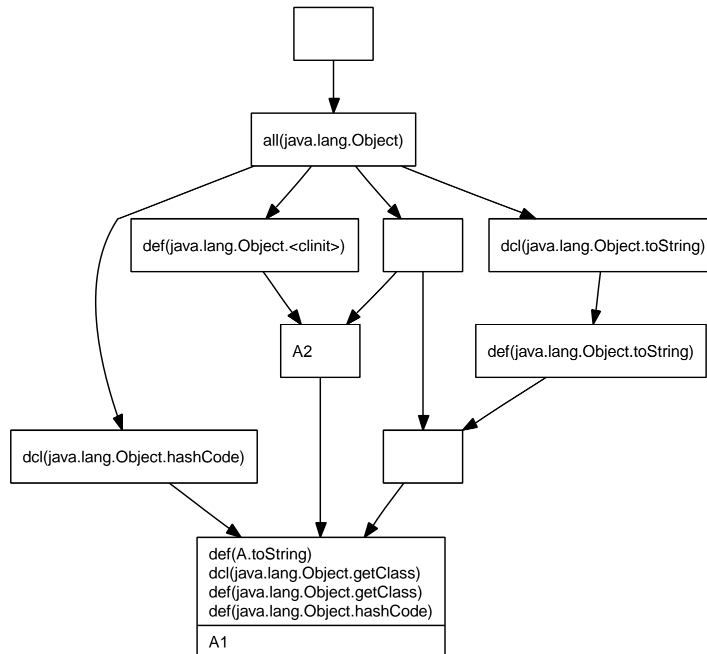


Figure 3.16: Class hierarchy for figure 3.14 with uncollapsed library classes

`A` objects access the constructor of object and get a table entry at the *all*(`java.lang.Object`) attribute. But this single attribute now *includes* the definition of `java.lang.Object.toString`, although this method is never accessed by the object. It has to be conservatively assumed, that if `toString` is called for $A2$, the implementation from `java.lang.Object` must be chosen. $A1$ gets an additional entry at the definition of `A.toString`, which will override the definition inherited from the class attribute.

Again, the result is perfectly valid, but the reduction of attributes brings new restrictions to modifying the class hierarchy later. But these restrictions do not seem acceptable for two reasons. (1) Not allowing two objects to have the same new type if they had the same original type is hard to sell to a user. (2) In addition to that it is dangerous, because the methods causing this will be crucial functions like `hashCode`. If these are left out from objects, which may not require them now, but may easily require them after further modifications of the program, it is far too easy to break the program. These problems do not seem to justify the win in performance archived through the modification.

Figure 3.16 shows the class hierarchy for the example above using individual attributes even for library classes[4]. The hierarchy is much more complicated, but it is also easy to see, that the second object of the program does not contain a definition for a `toString` method at all. This hierarchy can be transformed into one similar to figure 3.15 (for an algorithm see chapter 8.2), while then allowing the two `A` classes to be merged, as this time the information that $A2$ does not use a `toString` method is contained in the original class hierarchy.

But very obviously this version leaves the problem with the unwanted restructuring of library classes.

3.8.2 Recreation of Library Classes

Instead of forcing members of a library class to be at same node from the start, these nodes can be constructed after creation of the initial lattice. For each library class an equivalent class in the new hierarchy, that has the same members and inheritance relations as the original class, must be created.

While this does not sound too hard, care must be taken in detail, as members of library classes are usually scattered among many different classes and mixed with members from non-library classes (see figure 10.1 for a real world example).

As this algorithm works on the class hierarchy, it is described in full detail in section 8.2.

For each library class to be recreated, a *target* class must exists, where all attributes for that class are collated. As is it easy to move members up in

[4]The *empty* classes contain some pointers, which were omitted as they are not relevant here

the class hierarchy, a node above all members of a class can be constructed as follows:

$$\frac{Library(C)}{all(C) \in \mathcal{A}}$$

$$\frac{Library(C), def(C.m) \in \mathcal{A}C}{def(C.m) \rightarrow all(C)}$$

$$\frac{Library(C), dcl(C.m) \in \mathcal{A}C}{dcl(C.m) \rightarrow all(C)}$$

$$\frac{Library(C), C \leq C'}{\begin{array}{c} all(C) \rightarrow all(C') \\ r := all(C') < all(C), \\ r \in \mathcal{O}, \\ \bigwedge_{a}^{a \in \sigma(\{o|(o,all(C')) \in T\}) \setminus \{all(C)\}} (r,a) \in T \end{array}}$$

all(C) was already introduced in the previous section, it is the *class attribute* of C and create an infimum to all members of C.

The second and third constraints enforce the class attribute to be above all members of the its class. The fourth constraint preserves the original inheritance order and ensures that two class attributes will never be at the same node in the same way this was done for methods with equal signatures before.

All nodes with class attributes provide a kind of skeleton representing the original class hierarchy for library classes. The algorithm provided in section 8.2 will use this skeleton to recreate library classes in their original state.

3.8.3 Type-Constraints generated by Library Classes

Even if a pointer in the library is not used to access a member it will generated a type-constraint on a value passed to that pointer, as the type for the pointer itself must remain unchanged. Figure 3.17 shows an example of this. If the library class `LibraryA` is not treated specially, the new type for `Test.a1` and `Test.a2` would be $\top$ or `java.lang.Object`, as no members are accessed with these pointers. This breaks the static typing for the call to `id`, as the methods requires a parameter of static type `java.lang.String`. If the analysis forces the parameter of `id` to be `java.lang.String`, as it knows that `LibraryA` and thus the parameter of `id` will be immutable, `Test.a1` would be forced to be of type `java.lang.String` too, because an assignment between the two exists and then the static typing becomes right (The new type for `Test.a2` would still be $\top$, but that is not a problem).

```
1   class LibraryA {
2      static String id(String s) {
3         return s;
4      }
5   }
6
7   class Test {
8      static String a1;
9      static String a2;
10
11     public static void main(String[] args) {
12        a1=new String();
13        a2=LibraryA.id(a1);
14     }
15  }
```

Figure 3.17: Example program for type constraints created by library code

A pointer can be forced to have a certain library type c by creating a table entry at the class attribute $all(c)$, which was introduced in the previous section.

$$\frac{p \in ClassPtrVars(\mathcal{P}), Library(p)}{(p, all(type(p))) \in T}$$
$$\frac{p \in ClassVars(\mathcal{P}), Library(p)}{(p, all(type(p))) \in T}$$

This will ensure each pointer or object p has the required type to make assignments type-correct if p appears on the left side of the assignment.

For objects such table entries only becomes necessary, if empty or default constructors are ignored in the analysis. Otherwise, each object accesses a constructor of its class and this access will guarantee an object to be below $all(c)$.

Chapter 4

Points-To Analysis

For all kind of analyzes a static approximation of the objects a pointer can point to at run-time is interesting. As a result these kind of analyzes have been a research topic for years – a few years back mostly for C, recently also for Java. An overview can be found in [19]. In this chapter, the specific requirements of a points-to analysis for the Snelting/Tip analysis are presented. Then known algorithms are checked too see if they fulfill these requirements.

4.1 Background

This work was initially started in September 1998. Back than there were only a few publications about points-to analysis for Java, most work was still done for C. No "ready-to-use" analysis was available, so a new implementation had to be made. Goals for this implementation were set high (in no particular order):

(1) Modular architecture that would allow the plugging of any available points-to algorithm (with the main focus on Andersen's [3] and Steensgard's [44] analysis as they were well known). (2) Allowing enhancements like context-sensitivity or flow-sensitivity. (3) Whole program analysis, including library classes and native methods. (4) Support for reflection without forcing the user to manually specify possible targets. (5) Scalability for programs of 20000-100000 lines of code. (6) Soundness.

The final implementation does not fulfill some of these goals for various reasons. First of all, some established techniques for points-to analysis were not applicable for the specific needs of the Snelting/Tip analysis. This will be discussed in detail later in this chapter. Secondly, the influence of the library size on Java programs was hugely underestimated. Even small Java programs use an enormous amount of library code, very often the library code used has a multiple of the size of the analyzed program. Finally, other implementations have come up [37, 22], whose performance was significantly

better compared to the self-made implementation, so that it seemed unnecessary to search for further enhancements. For [22] it was evaluated in [7] whether it is usable to implement Snelting/Tip.

Still, the self-made implementation provides some advantages: (1) It is the only implementation known which is able to analyze at least some features of reflection without manual intervention by the user. This is described in detail in section 5.6. (2) It features a fall back conservative approximation that can be used if a part of the code is unavailable (e.g. a native method without a stub). (3) The analysis precision matches the specific needs of the Snelting/Tip analysis. Other points-to analyzes usually use different clients to evaluate their precision (Elimination of virtual method calls, synchronization removal and stack allocation are most commonly used), but this does not imply that Snelting/Tip will automatically benefit from their precision. (4) The analysis has been in use for some years now and its results have been checked countless times, providing no guarantee, but great trust in the correctness of the implementation.

4.2 Requirements

In the Snelting/Tip analysis, the points-to set is exclusively used to enumerate the objects that may be used as targets for a dynamic method call. That means if precision of a points-to analysis is evaluated, only the size of the points-to sets of points that are used for these calls are interesting. The set of those pointers is:

$$\begin{aligned}&\{p|p.m \text{ occurs in } \mathcal{P}, p \in \text{Pointers}(\mathcal{P}),\\ &\quad \exists C : \mathit{dcl}(C.m) \in \mathit{MemberDcls}(\mathcal{P}) \wedge \mathit{type}(p) \leq C\}\end{aligned}$$

Unfortunately, this specific requirement in precision is an unusual one. A similar and often used client of points-to analysis is the elimination of dynamic method calls. Here, the same set of pointers is relevant, but in this analysis a change in result is only achieved if for one call site the number of possible target methods can be reduced to 1, whereas in Snelting/Tip, every object removed from the points-to set makes a difference. As more efficient algorithms for the elimination of virtual method calls exist, points-to analysis is hardly used for that purpose in practice. So it provides only an artificial benchmark.

As strange as it sounds, a points-to analysis can be "too good" for Snelting/Tip. Figure 4.1 contains a small program fragment. A hypothetical, "smart" analysis can detect, that only the object $B1$ will be created, and that $\mathit{PointsTo}(p) = \{B1\}$. If that happens, $C1$ will not get a *def* entry and thus get a new type, that is not instantiable. But this analysis would be

```
...
class B {
   void f() { }
}
class C extends B {
   void f() { }
}
...
Object array[]=new Object[5];
B p;
if(array.length>3)
   p=new B(); // B1
else
   p=new C(); // C1
p.f();
...
```

Figure 4.1: Points-to example program

very useful to other points-to clients, e.g. it will detect, that the target of the method call $p.f()$ will always be the method $B.f$.

The reason that these results are not usable for Snelting/Tip is a mismatch in the set of assignments used within Snelting/Tip and inside the points-to analysis. If $\mathit{Assignments}(\mathcal{P})$ contains assignments that are not used during the points-to analysis, table entries for declarations may be propagated from a call site to objects, but they will not get corresponding entries for definitions, as the points-to analysis "knows" that these objects will not reach the call site.

It might be possible to retrieve the set of assignments from the points-to analysis. But if the points-to analysis omits assignments that are present in the source code, the resulting lattice will not preserve the semantics of the original program. This can be prevented by modifying the source code accordingly (e.g. removing the dead `else`-branch in the example above), but Snelting/Tip was not meant to modify programs at statement level and this should not be forced upon the user of the analysis.

The connection between $\mathit{Assignments}(\mathcal{P})$ and the points-to sets becomes very obvious with the compact definition of Andersen's points-to analysis within the description of the Snelting/Tip analysis (section 2.2.2).

4.3 Andersen for Java

The points-to algorithm used in this implementation is based on Anderson's algorithm for C[3] and extends it for Java where necessary. A similar ap-

```
r:=q
q:=p
p:=Q
q:=R
```

Figure 4.2: Example program

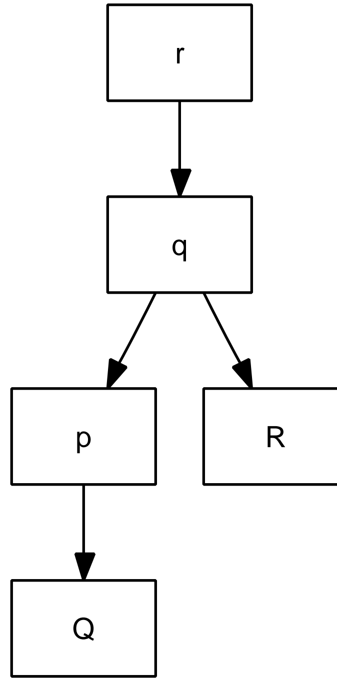


Figure 4.3: Storage graph for figure 4.2

proach [37] was developed in parallel, but predates an earlier publication ([35], [36], [46]). More about the differences later.

Anderson's algorithm processes all pointer related assignments of a program. For each assignment `p:=q`, it makes the points-to set of p a superset of the points-to set of q:

$$p := q \text{ occurs in } \mathcal{P} \Rightarrow \mathit{PointsTo}(q) \subseteq \mathit{PointsTo}(p)$$

Intuitively, this can be phrased as: p points at least to everything q points-to. The information gained from the assignments can be stored in a graph. Figure 4.2 shows an example program, where lower case letters represent pointers and upper case letters represent objects. A graph representation for the assignments can be seen in figure 4.3. Nodes in the graph is created for pointers and objects, and an assignment creates an edge from the left-hand to the right-hand side.

To retrieve the points-to set for a pointer p, the graph can be traversed by depth-first or breadth-first search starting at p, creating a list of all object nodes, which is the points-to set of p.

Retrieval of a single points-to set has complexity $O(n^2)$, making the algorithm $O(n^3)$ if all points-to sets must be retrieved. It is possible to add

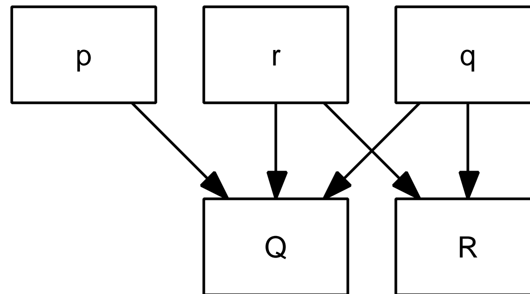


Figure 4.4: Alternate points-to graph for figure 4.2

all transitive edges to the graph in $O(n^3)$ and after that, retrieval is only $O(n)$. Depending on the graph implementation, the second version may use significantly more memory, as the graph will contain a lot more edges. Space requirement for the storage of points-to sets is important. [22] compared different approaches for the storage of points-to sets. There is a recent trend to store points-to sets as binary decision diagrams [5, 56]. For Snelting/Tip the first version helps saving a lot of memory, as it automatically includes the set $Assignments(P)$.

To handle dynamic method calls with Andersen, a fix-point iteration is required. First an initial graph from only the assignments is created from the code. Then, if a method call $p.f(...)$ occurs within the program, $PointsTo(p)$ is calculated and assignments added for the `this`-pointer, return values and parameters of f. This is done as a fix-point iteration, until $PointsTo(p)$ is stable.

Many of the other aspects where Java is different from C are handled in a similar way. All conditional constraints presented in appendix A are basically part of the fix-point loop.

As said before, [37] provides a adaption of Andersen for Java that is very similar to this work. They use Bane, a generic constraint solving system [10, 14, 48], that has been used for pointer-analysis before [12]. This implementation is generally faster and requires less memory, compared to the approach described here. While it is not surprising that a specialized constraint solving system outperforms an unoptimized self-made system, there is a difference in the way the points-to graphs are created.

For an assignment $p = q$, where p and q are pointers, they do not create edges between the nodes for p and q, but edges between p and the elements of $PointsTo(q)$. In this representation, the graph for the example above can be seen in figure 4.4. This graph can be constructed from our graph by adding the transitive hull and removing all edges from pointers to pointers.

A second difference is the handling of fields. The example in figure 4.5 is taken from [37]. In this version for each field and object, a node

representing this field is created (figure 4.6(b)), while they annotate edges with field names (figure 4.6(a)).

4.4 Steensgaard for Java

Another well known points-to analysis is Steensgaard's[44]. The basic idea here is to use unification for assignments, i.e. for an assignment `p=q` it is assumed, that the points-to sets of p and q are equal. While this is less precise than Andersen's algorithm, it is significantly faster. Because every assignment needs to be processed only once, the complexity is quasi linear.

The analysis was written for C and applying it to Java causes some problems. First of all, the `this`-pointer of the constructor of `java.lang.Object` points to every object in the program and this means via unification, all pointers point to all objects. This can be solved by ignoring the constructor of object, as it is empty, or by using context-sensitivity.

The other issue is the handling of dynamic bound method calls. If the target is resolved by using the previous points-to results, the analysis would loose it linear complexity. According to Steensgaard[45], the logical extension to his analysis for Java is to merge the `this`-pointers for the called methods.

Unfortunately this extension does not work for Snelting/Tip, as it deliberately throws away precision at exactly the point, where Snelting/Tip needs it. The result is an identical access pattern for all objects of the same original type, disabling the detecting of individual access patterns (which was the purpose of Snelting/Tip) completely. Therefore it is not reasonable to use this analysis for Snelting/Tip.

4.5 Flow-Sensitivity

The original Snelting/Tip paper assumed that all possible points-to algorithms can be used, as no further assumptions about the nature of the points-to sets were made. But then, the expected format of the results of the points-to analysis only fits to flow- and context-insensitive analyzes. Still, these are established techniques in the area of points-to analysis and their suitability for Snelting/Tip will be examined.

In a *flow-sensitive* analysis the order of the statements in a program is taken into account, while a *flow-insensitive* analysis ignores it. As as result, the calculated points-sets are no longer valid for an entire program, but only at a certain place within the program. For the following example program:

```
1 p:=Q;
2 q:=p;
3 p:=R;
```

```
p=Q;
q=R;
p.f=q;
r=r.f;
```

Figure 4.5: Example program for field handling

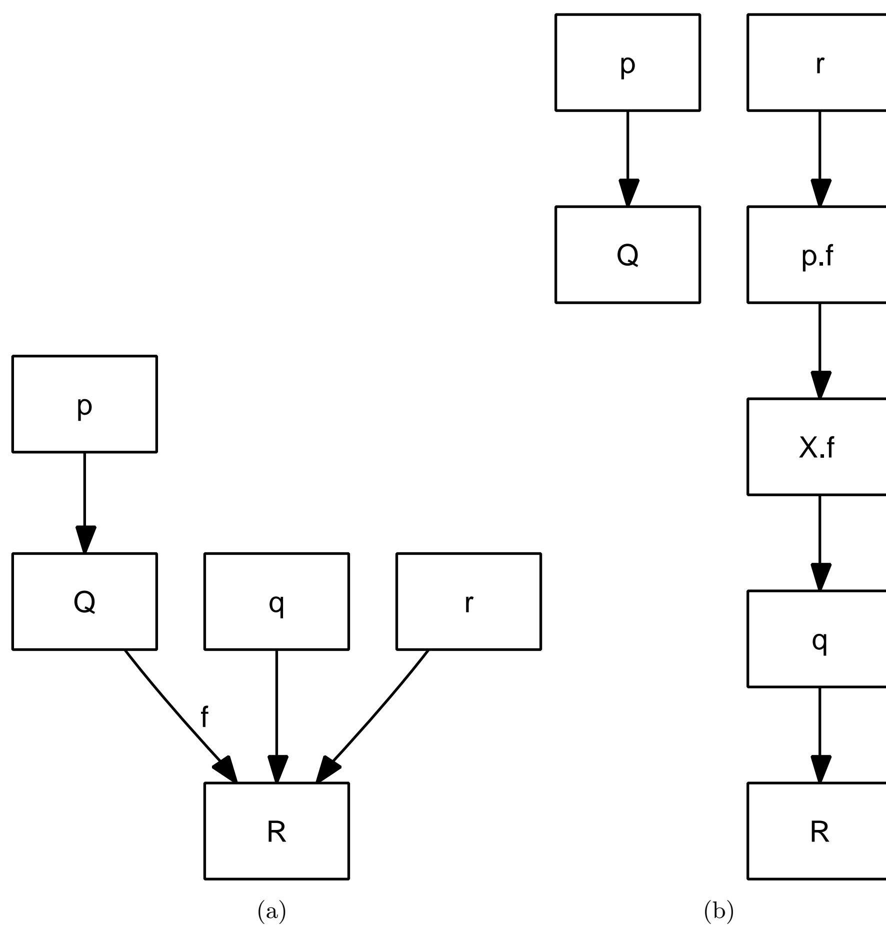


Figure 4.6: Points-to graphs for the program in figure 4.5

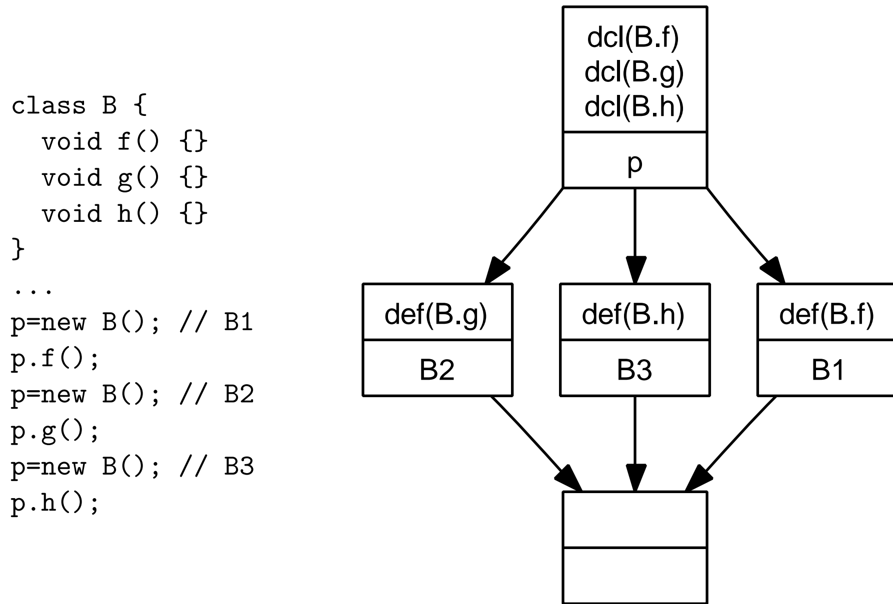


Figure 4.7: Points-to example program and class hierarchy

the points-to sets by a flow-sensitive analysis are as follows:

$$\begin{aligned} PointsTo(p, 1) &= \{Q\} \\ PointsTo(p, 2) &= \{Q\} \\ PointsTo(q, 2) &= \{Q\} \\ PointsTo(p, 3) &= \{R\} \\ PointsTo(q, 3) &= \{Q\} \end{aligned}$$

where $PointsTo(v, n)$ denotes the points-to set of variable v after the execution of program line n. A flow insensitive-analysis will produce these sets:

$$\begin{aligned} PointsTo(p) &= \{Q, R\} \\ PointsTo(q) &= \{Q, R\} \end{aligned}$$

The difference in precision is obvious. Flow-sensitive analysis is usually better than flow-insensitive analysis, but never worse.

The Snelting/Tip analysis can be easily adapted to use the results from a flow sensitive analysis. The points-to sets are only used to resolve method calls and instead of using a global points-to set, the site specific set can be used. Figure 4.7 shows an example program and the resulting lattice. Here, for the method call $p.f()$, the points-to set for p would contain only $B1$, for $p.g()$ it contains $B2$ and for $p.h()$ only $B3$. The resulting lattice shows that none of the new types for the objects can be instantiated, as all of them miss the implementation for two methods.

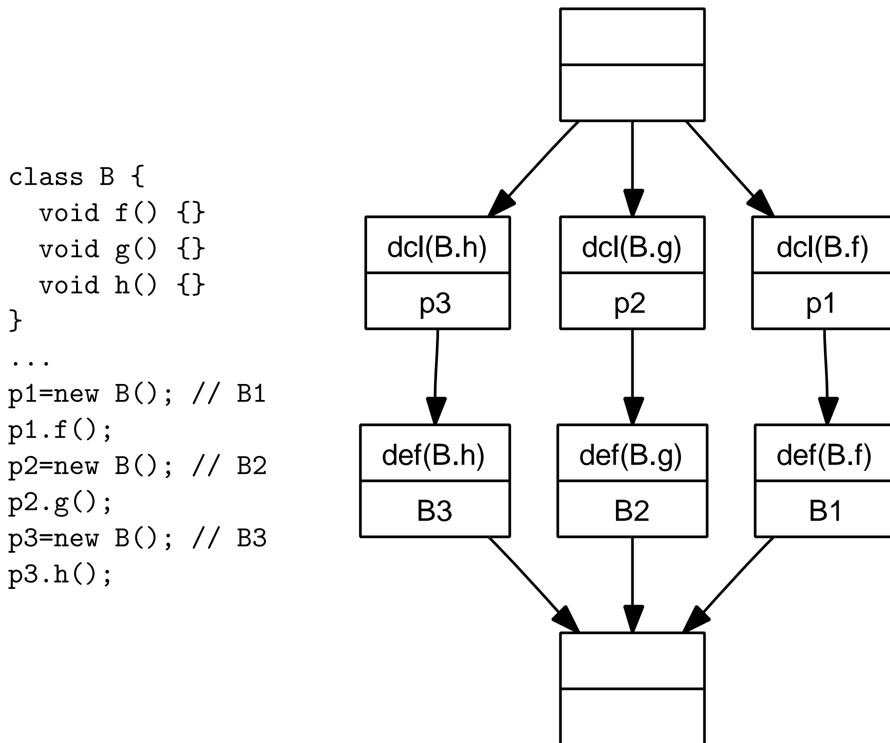


Figure 4.8: Program from figure 4.7 in SSA form

A similar effect to intra-procedural flow-sensitivity can be achieved by transforming the program into single static assignment form[18]. The transformed example and the resulting lattice can be seen in figure 4.8. In this version the lattice is fine, but as this requires a statement level transformation too, it might be unwanted by a user of Snelting/Tip.

Another possibility is to analyze a program flow-sensitively and later merge the information for the same variable from different program points into a single points-to set. This will result in the same precision as a non-flow sensitive approach, but might be useful if flow-sensitive analysis is cheaper than a flow-insensitive analysis. But this is not true in general, so this remains a hypothetical optimization.

4.6 Context-Sensitivity

A points-to analysis is called *context-sensitive* if the analysis takes a call site (calling context) to a method into account while analyzing this method. This increases precision but also the cost in time and space for the analysis, as a method may be analyzed multiple times and analysis results for that method must be store multiple times.

This is especially useful for an object-oriented language, where get- and

```
class A {
   void f() {}
   void g() {}
}
class B {
   A a;
   void set(A _a) { a=_a; }
   A get() { return a; }
}
...
B b1=new B(); // B1
b1.set(new A()); // A1, [1]
A a1=b1.get(); // [2]
a1.f();
B b2=new B(); // B2
b2.set(new A()); // A2, [3]
A a2=b2.get(); // [4]
a2.g();
```

Figure 4.9: An example with get- and set-methods

set-methods for a data-members of an object are common. Without special handling, these structures make a points-to analysis imprecise.

Figure 4.9 shows a program fragment, which include a class with get- and set-methods and their usage. This is a very common pattern in object-oriented programs, in the simple form as well as in the more complex form of container classes.

The points-to graph can be seen in figure 4.10. The points-to sets from this graph are $PointsTo(a1) = \{A1, A2\}$ and $PointsTo(a2) = \{A1, A2\}$. This is imprecise, as in practice each variable will point to only one object. The points-to graph also reveals the source of the imprecision. While there are individual storage locations for $B1.a$ and $B2.a$, the passing of $A1$ and $A2$ to the `set` method and retrieving it with the `get` method causes the imprecision.

This imprecision can be avoided, if all calls to `B.set` and `B.get` are treated context-sensitively. A points-to graph where this has been done can be seen in figure 4.11. Contexts are denoted by `[1]`, `[2]`, `[3]` and `[4]` as prefixes to the variables that are specific for the methods analyzed within this context. The context corresponds to the call sites within the program. Handling methods context-sensitively results in a multiplication of the nodes for variables of these methods, but exactly this increases precision. In this version, the better results $PointsTo(a1) = \{A1\}$ and $PointsTo(a2) = \{A2\}$ can be determined because there are individual pointers for parameters,

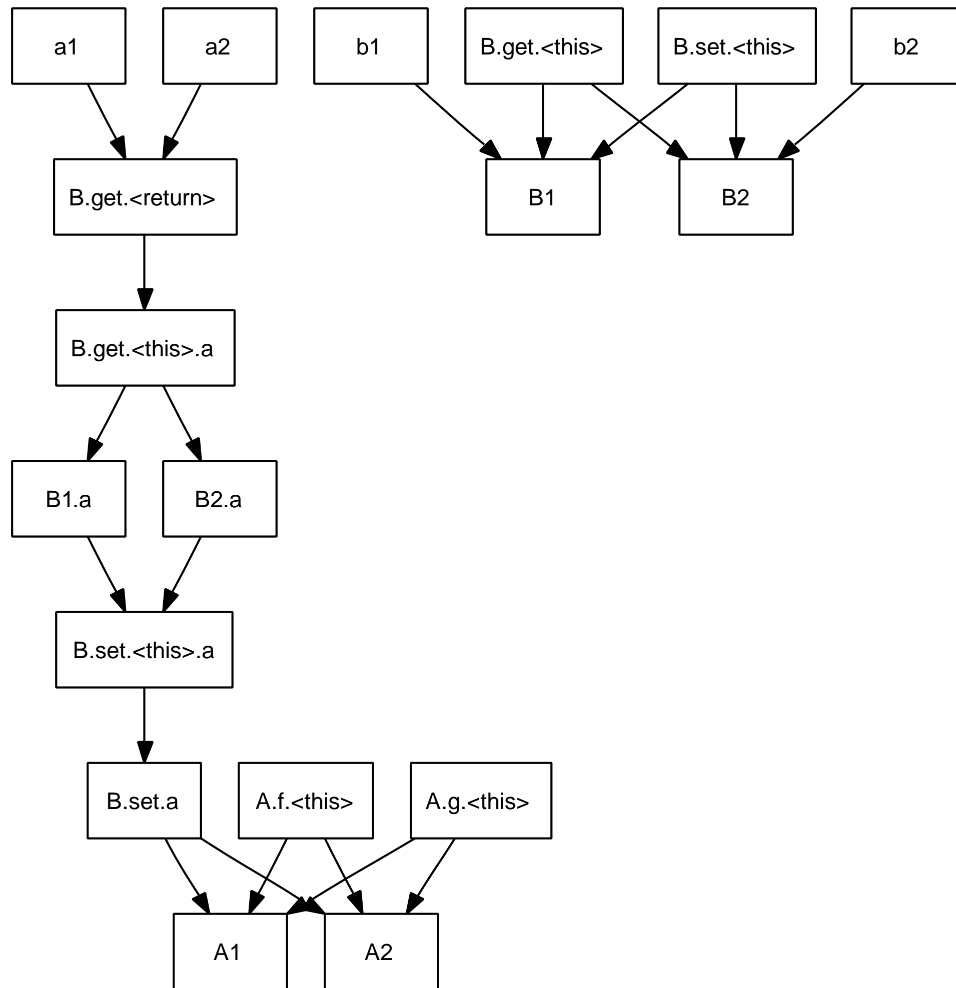


Figure 4.10: Points-to graph for figure 4.9

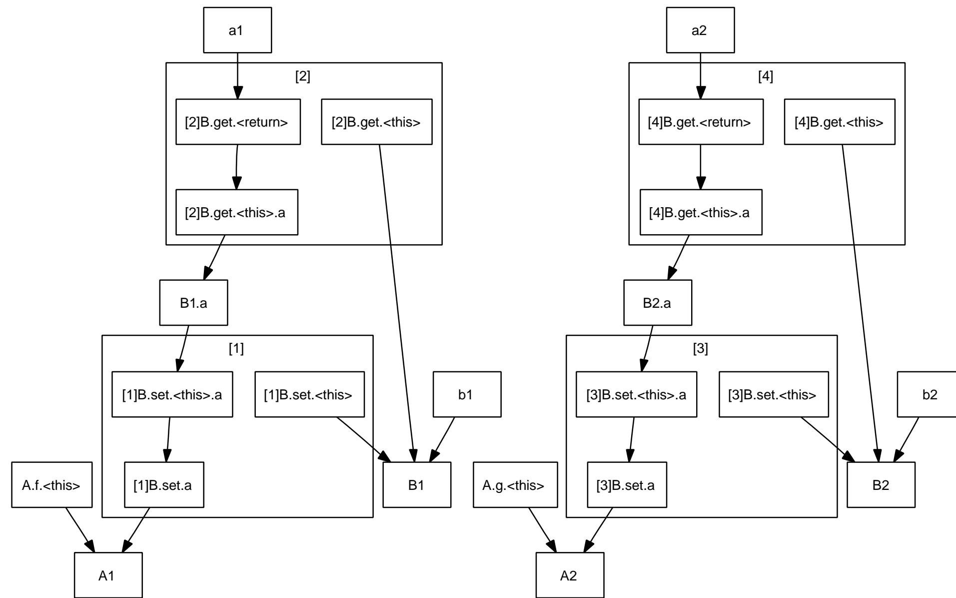


Figure 4.11: Context-sensitive points-to graph for figure 4.9

`this`-pointer and return values of each `get` and `set` call.

But how will points-to results from a context-sensitive analysis influence the Snelting/Tip analysis? Figure 4.12 shows an example program and the concept lattice, where the factory method `A.create` is analyzed individually for each call site.

The resulting class hierarchy is valid, but it suggests two different types for the object $A1$. This can be realized, if the code for `create` is put multiple times into a transformed program: once into the class containing f and once into the class containing g. While this may look fine for this example, it is not a generally usable approach, because the code size would explode with an increased number of contexts and even duplication of a single method can be considered questionable.

In many cases, a points-to analysis that is more precise also gets faster, because less objects have to be collected or propagated. Even if Snelting/Tip cannot profit from the better results of context-sensitive analysis, it might still be possible to do a context-sensitive analysis, thereby using the increased precision to speed up the analysis and later simply "throw away" the increased precision and still benefit from the decreased analysis time.

In the example above, the main problem was caused by a variable appearing in different contexts, which is a problem for the Snelting/Tip analysis as this information cannot be mapped back into source code. If a variable appears in different contexts, all instances of the same variable must be merged into one variable to avoid this. This merging can be easily realized

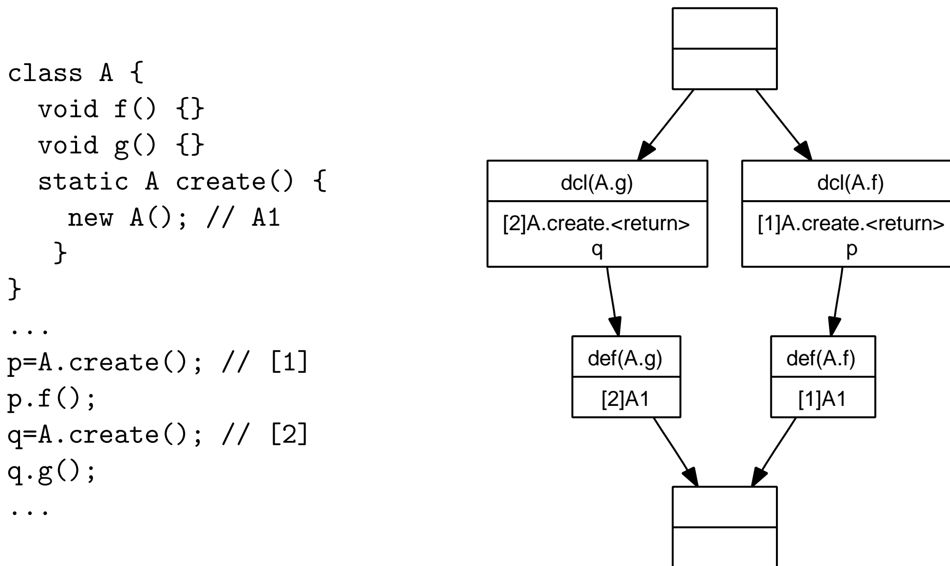


Figure 4.12: Example for context-sensitivity and Snelting/Tip

by melting the variable nodes in the points-to graph. As this operation may result in a lot of new points-to relations, the usual propagation must be done at this stage in order to check if virtual method calls are reached by new objects which then in turn my result in new assignments. However, the cost for this propagation is the same cost, that would be required for a context-insensitive analysis, as the resulting points-to graph for both analyzes will be identical. So a lower analysis cost is only gained, if this propagation can be omitted.

Unfortunately, a simple example is sufficient to show that this will result in incorrect class hierarchies. Figure 4.13 shows the result of this merging. This points-to graph is almost identical to the original graph in figure 4.10, but keeps the increased precision for the `this`-pointers of $A.f$ and $A.g$. But the resulting class hierarchy will again be unusable, as both objects $A1$ and $A2$ will get entries $dcl(A.f)$ and $dcl(A.g)$, but only a def entry for one of the two methods, leading to non-instantiable classes as seen before.

While context-sensitivity is a well known and valid way to enhance points-to analysis, it is not usable for Snelting/Tip as a client analysis.

4.7 Object-Sensitivity

Instead of using the call-site as a context, the object used for the method call can be used as a context[27]. A points-to graph for the example from the previous section can be seen in figure 4.14. Here `[B1]` and `[B2]` are used as context names, as $B1$ and $B2$ are the objects used to dispatch the calls to the methods `get` and `set`.

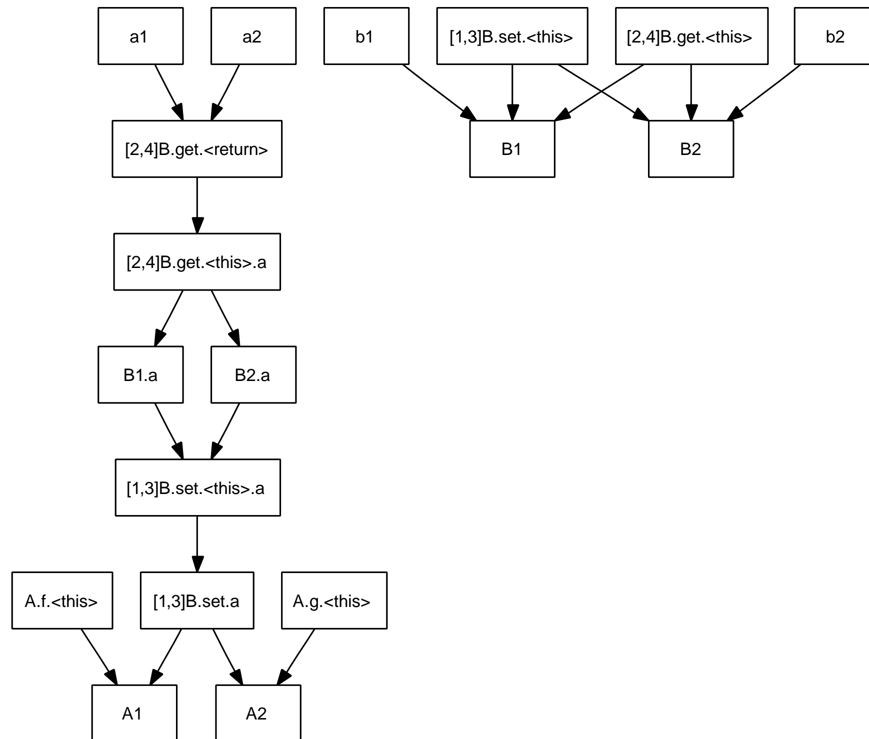


Figure 4.13: Figure 4.11 with contexts merged

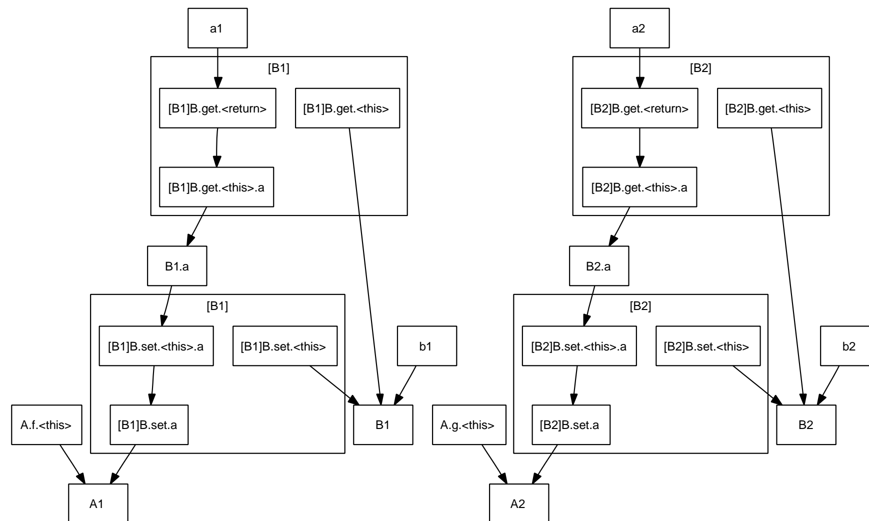


Figure 4.14: Object-sensitive points-to graph of 4.9

It is easy to see from the example that object-sensitivity gives the same benefit in terms of precision as context-sensitivity, but while the cost (measured in the number of used contexts) is the same for this example, this is not the general case. Object-sensitivity is cheaper, if a lot of calls are made to the same container objects, while context-sensitivity is cheaper if a small number of call sites is reached by a huge amount of objects. E.g. for the given example, object-sensitivity will be become more expensive, if it is possible for $b1$ and $b2$ to point to multiple B objects instead of only one, while context-sensitivity will be more expensive if more calls to `set` or `get` are added.

Observation of existing Java programs and the Java class hierarchy makes object-sensitivity seem more appropriate for Java. The internal call graph for some of the standard containers is highly complex, leading to high costs for context-sensitivity. Which method is best for a particular program must be decided individually. Both methods can even be combined.

For Snelting/Tip object-sensitivity provides the same problem as context-sensitivity: an object or pointer may get different types in different contexts, which cannot be realized in a Java program. The same problems with merging arise too, making also object-sensitivity unusable for Snelting/Tip.

4.8 Other Points-To Analyzes for Java

There are a lot of other points-to analyzes for Java [57, 11, 23]. Whether they are suitable for use with Snelting/Tip is often not clear from these papers. Most of them miss enough details for any of following topics: handling of arrays, handling of explicit and implicit exceptions, handling of library classes, handling of native methods or handling of type casts. Implementations are only rarely available and re-implementing each analysis might be a vain endeavor because of the risk of detecting it does not work well with Snelting/Tip. The results with well-known techniques like context-sensitivity are not very encouraging.

One system is freely available and an implementation of Snelting/Tip using SOOT and SPARK was done by Buckley [7]. His results can be summed up as follows: Although SPARK is also described as an extension of Andersen for Java, the analysis is less precise than [37] and the work presented here. The imprecision explicitly affects `this`-pointers and makes SPARK uninteresting for the Snelting/Tip. Buckley suggested an improvement that enhances the precision for `this`-pointers to make the analysis suitable for Java. However, some issues remain. SOOT does not model exception objects implicitly created by the JVM (e.g. the `NullPointerException` that is thrown for an expression $a.x$ the reference a is `null`). Omission of these objects will affect correctness of a transformed program, if a program relies

on presence and catching of these exceptions. Another issue is the handling of native methods, which has not been examined in detail.

[19] contains the statement that points-to analysis should be client specific. While the self-made implementation does not scale as good as other peoples implementation, it fits the specific requirements of Snelting/Tip best.

Chapter 5

An Analysis of Java-Bytecode

This chapter describes an analysis of Java-bytecode that will extract all required information for the Snelting/Tip analysis from a given program.

Java-bytecode was chosen as a bytecode parser is easier to implement than a source code parser. It has also been more stable than Java itself: bytecode has only undergone minor changes from JDK 1.0 to 1.5, while Java itself was changed heavily (e.g. inner classes, generics).

5.1 Introduction

To implement the Snelting/Tip analysis, member accesses and assignments must be extracted from the bytecode and a points-to analysis must be performed. Bytecode is stack-oriented, i.e. most instructions fetch their arguments from the stack and put their result onto the stack. In addition, there is method-local random access memory in form of *local variables*. Special instruction transfer information from the stack into these variables and vice versa. Bytecode instructions may have additional operands. These operands are fixed and cannot be changed during run-time.

E.g. for the following Java fragment:

```
...
   x=new Y(); // Y1
   y=x;
   y.f=z;
...
```

a compiler can produce the following bytecode:

```
...
 9   new Y    ;; create new object and put reference on stack
```

```
10   astore 0 ;; store top stack content in local variable 0
11   aload 0  ;; put content of local variable 0 onto stack
12   astore 1
13   aload 1
14   aload 2
15   putfield Y.f
...
```

Additionally, the compiler creates a local variable table:

start_pc	length	name	type	index
11	10	x	Y	0
13	10	y	Y	1
1	20	z	java.lang.Object	2

In the example, the `aload` and `astore` instructions transfer data between stack and local variables, while the `putfield` instruction writes a field of an object. This is one of the member accesses required for Snelting/Tip. The instruction takes two values from the stack. The topmost value is written to the modified field (i.e. it corresponds to `z` to the Java program) and the second value is the object reference whose field `f` will be written (`y` in the original program). This reference is required to determine all accesses to the field `f`.

Looking at the example in detail reveals that the value for `y` was originally put onto the stack by instruction 13, which read it from local variable 1. It was put into this variable by instruction 12, which took it from the stack, where is was put by instruction 11, which copied it from local variable 0, and ...

This kind of *tracing* a value backwards on the stack can be formalized by representing the stack and local variables as sets of possible values they can have at run-time. This must be done individually for each point in the control-flow of the method. The bytecode instructions can than be expressed as a relation between these sets before and after an instruction's execution.

Let $S(i, n)$ be a set which contains all possible values of the n-th stack element before the instruction at program counter i is executed and $R(i, n)$ the content of the local variable n before instruction i.

The constraints generated from the bytecode instructions of the example are (assuming the stack was empty before instruction 9):

$$\begin{array}{lr} \texttt{(9)} & \{Y1\} \subseteq S(10,0) \\ \texttt{(10)} & S(10,0) \subseteq R(11,0) \\ \texttt{(11)} & R(11,0) \subseteq S(12,0) \\ \texttt{(12)} & S(12,0) \subseteq R(13,1) \\ \texttt{(13)} & R(13,1) \subseteq S(14,0) \\ \texttt{(14)} & R(14,2) \subseteq S(15,0) \end{array}$$

Of course if a local variable or a part of the stack is not modified by an instruction, it must remains unchanged, so additional constraints for this data flow are required:

$$
\begin{array}{ll}
\texttt{(9)} & R(9,0)\subseteq R(10,0) \\
 & R(9,1)\subseteq R(10,1) \\
 & R(9,2)\subseteq R(10,2) \\
\texttt{(10)} & R(10,1)\subseteq R(11,1) \\
 & R(10,2)\subseteq R(11,2) \\
\texttt{(11)} & R(11,0)\subseteq R(12,0) \\
 & R(11,2)\subseteq R(12,2) \\
\texttt{(12)} & R(12,0)\subseteq R(13,0) \\
 & R(12,2)\subseteq R(13,2) \\
\texttt{(13)} & R(13,0)\subseteq R(14,0) \\
 & R(13,1)\subseteq R(14,1) \\
 & R(13,2)\subseteq R(14,2) \\
\texttt{(14)} & R(14,0)\subseteq R(15,0) \\
 & R(14,1)\subseteq R(15,1) \\
 & R(14,2)\subseteq R(15,2) \\
 & S(14,0)\subseteq S(15,1) \\
\texttt{(15)} & R(15,0)\subseteq R(16,0) \\
 & R(15,1)\subseteq R(16,1) \\
 & R(15,2)\subseteq R(16,2)
\end{array}
$$

Subset constraints must be chosen instead of equality, as in general control-flow is not linear like in the given fragment. E.g. instruction 15 could be the target of a jump instruction later in the program, making $S(15,1)$ different from $S(14,0)$.

These sets can be interpreted as rows for the Snelting/Tip analysis, because in bytecode member access actually happens on these sets (e.g. instruction 15 would create a table entry $(S(15,1), \mathit{def}(Y.f))$, but this is a much too fine grained view, as it would create many unused table rows. Usually, table rows are created for pointers and objects of the program. The pointers are available in the local variable table. Each entry of this table can be interpreted as a *meta-set* of local variable sets, as it represents multiple sets $R(i,n)$ where n is the *index* column of the local variable table and i the range $(\mathit{start_pc}, \mathit{start_pc} + \mathit{length})$.

The meta-sets for the given example are:

$$
\begin{aligned}
x &= \{(R(11,0), \ldots, R(20,0)\} \\
y &= \{(R(13,1), \ldots, R(22,1)\} \\
z &= \{(R(1,2), \ldots, R(20,2)\}
\end{aligned}
$$

Then, each set can be traced to a table row by the following algorithm:

```
getrow'(S, been_at):=
  if ∃m_1, ..., m_n ∈ MetaSets : n ≥ 1  ∧  S ∈ m_1  ∧ ... ∧  S ∈ m_n
     return {m_1, ..., m_n}
  if ∃S_1, ..., S_n ∈ MetaSets : n ≥ 1  ∧  S_1 ⊆ S  ∧ ... ∧  S_n ⊆ S
     return ⋃_{i=1}^{n} getrow'(S_i, been_at ∪ {S})
  return S
getrow(S):=
   return getrow'(S, {})
```

The algorithm first checks if the current set is contained in any meta-set and returns those. If it is not, all sets connected via sub-set constraints are traversed recursively. If no subset constraints exists, the set must be an initial set created by a `new` instruction and contains an object. This object is then returned.

For the given example, $S(15, 1)$ is a good candidate to resolve, as it is used for member access. As $S(15, 1)$ is not contained in any meta-set, all subsets will be visited, i.e. $S(14, 0)$. $S(14, 0)$ is not contained in any meta-set either, so all subsets of it ($R(13, 1)$) are visited. $R(13, 1)$ is contained in the meta-set y, so $getrow(S(15, 1)) = \{y\}$, which is exactly the variable used for the member access in the Java program before compilation.

Please note that the meta-sets are an additional input and may be used to adjust the number of table rows actually present. Without definition of any meta-set, $getrow(S(15, 1))$ will be directly resolved to $Y1$, which is a valid result for the given program.

With the help of the *getrow* function, assignments can be easily gained from the constraint system:

$$X \subseteq Y \rightarrow (getrow(Y), getrow(X)) \in Assignments(\mathcal{P})$$

The constraint system implicitly contains Andersen's points-to analysis: the content of a set $S(x, y)$ already is its points-to set, and for a meta-set, it can be easily calculated by merging all contained sets:

$$PointsTo(x) = \bigcup_{y \in x} y$$

Therefore the constraint-system provides all information needed for the Snelting/Tip analysis.

Creation of such a constraint-system is only possible due to some guaranteed structural properties of Java-bytecode:

- The depth of the stack before each instruction can be statically determined and is invariant. This implies that the maximum stack size is finite and statically known.

- The number of a local variable for each access is known statically. This implies that the maximum number of a local variable is known.
- The set of instructions following another instructions in the control-flow is statically known.

5.2 Intra-Procedural Analysis

The intra-procedural analysis handles one method at a time. The analyzed method will further be called M and used like a free variable. A method name M includes the class name and signature, making it an unique identifier for a method.

In addition to M a second free variable, a context C will be used throughout the analysis. C can be used to make the analysis context- or object-sensitive by parametrization[1]. For the intra-procedural analysis no further knowledge about this context is required and as a value for C a context $EMPTY$, which represents the absence of any contextual information, can be assumed.

The analysis builds a constraint system around possible contents of local variables and stack of the JVM at run-time. These contents will further be called *values*. Three categories of values exists:

1. base types
2. references
3. return addresses

The base types are a simplified version of the non-object types of the Java language. Special types for `boolean`, `byte`, `char` or `short` do not exist, `int` is used instead. Actual values of these are ignored in this analysis, so these are always represented by their type-name: `<double>`, `<float>`, `<int>`, `<long>`.

Return addresses are used to allow local subroutines within a single byte-code method. These return addresses are natural numbers and are handled individually, as their concrete values are known and usually there are only a few of them for each method. Handling the actual values provides a small increase in analysis precision at small cost, as it can be used remove control-flow that will never happen at run-time.

Each value v has a type. Figure 5.1 defines these types for all kind of values as well as some additional functions used in the constraints. The $\leq$

[1] Although these techniques are not usable for Snelting/Tip, contexts are left in the formalism, because the analysis itself is not limited to Snelting/Tip, but could be used for other purposes, where context- and object-sensitivity are more useful.

$$type(v) = \begin{cases} v & \text{if } v \text{ is an abstraction of a base-type (e.g. } \texttt{<int>}\text{)} \\ c & \text{if } v \text{ is a reference and } c \text{ is the class of that reference} \\ \texttt{<ra>} & \text{if } v \text{ is an return address} \end{cases}$$

$$type(v) \leq type(v') \Leftrightarrow \begin{array}{l} v \text{ and } v' \text{ are references } \wedge \\ v \text{ is assignment-compatible to } v' \end{array}$$

$$category(t) = \begin{cases} 2 & \text{if } v = \texttt{<double>} \vee v = \texttt{<long>} \\ 1 & \text{else} \end{cases}$$

Figure 5.1: Definition of types and associated functions

operator refers to assignment compatibility ([24, 2.6.7]) rather than the subtype relation in Java, as assignment compatibility includes rules for array-types which are not fully integrated into Java's type system. The *category* function maps types to the category they have in the JVM. Its result refers to the number of local variables a value of that type occupies and is also used in some stack operations.

Bytecode and its semantics is defined by [24] in its binary representation as `.class` files. Each class is defined by one such file, which contains all structural information about the class as well as the actual bytecode for all methods of that class. Although it might be possible to generate a constraint system directly from the binary representation, this would result in very cryptic rules that are hardly human readable. So instead of the binary representation, an assembler like notation of bytecode is used, similar to the output of `javap -c` or `jas`[2]. A formal definition of this notation is not given, but by using [24] it is obvious how to gain the notation used here from the binary representation.

The code of a single method in bytecode consists of one or more instructions. Each instruction is identified by an individual opcode which is stored first in the binary representation and zero or more following operands. The number of operands is fixed for most operands, with two exceptions where the total number of operands depends on the first operand. In any case, the number of operands does not depend on information which is not statically known. Each bytecode instruction has an address, which is the offset from the start of the bytecode to the start of the instruction. So the first instruction of a method has always the address 0 and the second will be equal to the length of the first instruction and so on.

The analysis generates constraints for each bytecode instructions. The constraints depend on the address of the bytecode instruction (denominated

[2] `jas` is a wide-spread Java assembler.

$PARAMS(M)$	The number of parameters of method M
$TPARAM(M, n)$	The type of a value of the n-th parameter of method M
$TRETURN(M)$	The type of the return value of method M
$LENGTH(M)$	The length of the bytecode for method M in bytes
$EH_MAX(M)$	The number of exception handlers in M
$EH_START(M, n)$	The start address of exception handler n
$EH_END(M, n)$	The end address of exception handler n
$EH_TYPE(M, n)$	The caught type or NULL of exception handler n. In case of NULL the handler catches every exception.
$EH_PC(M, n)$	The target address of the exception handler n

Figure 5.2: Bytecode properties used in constraints

pc) and the operands of the instruction (if any) or properties of the analyzed method or the class hierarchy in general. Table 5.2 contains some symbols that are used to refer to these properties within the constraints. Although they may look like function calls, these are static properties which can be derived from bytecode easily.

The sets used in the constraints primarily model the local variables and stack content of the JVM while executing the methods. Additional sets are needed to model data-flow from and to other methods. The requirement for parameters and return values are obvious, but the transitively reachable data through fields and array contents is also important. In contrast to the former sets, these are not dependent on the analyzed method, but global sets shared by all methods. As Java has no multi-dimensional arrays but uses nested one-dimensional arrays instead, the handling for fields of objects and content of arrays is almost identical. Table 5.3 lists all sets used in the constraints. The verification process guarantees that the sets modeling stacks and local variables will never mix content of different base types or base types and return addresses. Any preconditions which are always fulfilled due to verification are omitted from the constraints.

The actual generation of the constraints from bytecode is done by the algorithm given in figure 5.4. The algorithm has two parts. The first part creates the initial content for the local variables from the parameter sets and the second part processes all instructions in the bytecode and generates the corresponding constraints.

The constraints for all instructions are listed in appendix A. For most instructions, the constraints are obvious and easy to understand. However, for some of them, some words of explanation seems helpful.

The instructions `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2` have multiple cases

$R(M, C, pc, n)$	The content of local variable n before execution of instruction of pc
$S(M, C, pc, d)$	The content of the stack at depth d before execution of instruction pc
$\mathit{THIS}(M, C)$	The `this`-pointer of method M. For static methods, the set will be empty.
$\mathit{PARAM}(M, C, n)$	The n-th parameter of method M (starting with n=1).
$\mathit{RETURN}(M, C)$	The values returned by method M.
$\mathit{EXCEPTIONS}(M, C)$	The exceptions thrown by method M, includes exceptions that are thrown in methods called from within M and not caught by any handler in M.
$\mathit{CONTENT}(o)$	A set containing all values stored as content of the array o.
$\mathit{FIELD}(o, f)$	A set containing all values assigned to the field f of object o.

Figure 5.3: Sets used in constraints

$next$ `:=0`
`if` not $static(M)$
    add constraint $\mathit{THIS}(M, C) \subseteq R(M, C, 0, 0)$
    $next := 1$
$\forall i = 1 \ldots \mathit{PARAMS}(M)$:
   add constraint $\mathit{PARAM}(M, C, i) \subseteq R(M, C, next, 0)$
   $next := next + category(type(\mathit{PARAM}(M, C, i)))$

$pc := 0$
`while` $pc \neq \mathit{LENGTH}(M)$
    decode instruction at pc
    add constraints for instruction at pc
    $pc := pc +$ length of instruction at pc

Figure 5.4: Pseudo-code algorithm for constraint generation from bytecode

depending on the type of operands on the stack. Which variant is used, can be determined statically and for each instruction only one variant will be used. The multiple cases can be avoided by handling the types LONG and DOUBLE as types allocating two elements on the stack. This notation was used in the first edition of the JVM specification[24], but changed in the second. For generation of constraints the old notation would make the `dup*` instructions easier, but complicate other constraints for other instructions, especially method calls.

`jsr` and `jsr_w` push the address of the following instruction on the stack which can later be used by an `ret` instruction. As these addresses are statically known, they are handled with their actual value. This reduces the number of instructions possibly following a `ret` instruction, as otherwise all instructions following `jsr` instructions have to be assumed to be possible succeeding instructions to every `ret`, creating a lot of potential data-flow which cannot happen at run-time.

All `invoke*` instructions have to be distinguished between value returning and void methods, as they operate differently on the stack.

Various instructions create additional *rules*. These rules are used for two reasons. First, they factor out common constraints that are used by multiple instructions and thus make the constraints more human-readable and secondly, they allow a change of their definition, making it easier to modify the analysis later. Rules are defined globally, so all variables they use must be in their parameter list and unbound variables within a rule are not allowed. These parameters are instanced with actual values when a rule is made *valid* by a constraint. If this happens, the body of the rule definition is added to the constraint system and formal parameters are replaced by their actual values. A basic definition of the rules can be found in figure 5.5.

Some instruction generate a rule called *INIT*, which handles the initialization of referenced classes. Besides the exceptions which may be generated by this operation, a method called `<clinit>`, which initializes all static members of that class. This call uses a non-standard version of exception handling, as all non-checked exceptions are wrapped into exceptions of class `ExceptionInInitializerError`. The *INIT* rule handles this special case.

THROW is used to handle an exception. The last parameter n indicates, which exception handler tries to catch that exception. If the exception is not caught by a handler, another rule with $n + 1$ becomes active. If it is caught, control flow is branched from the current instruction to that exception handler. Finally, if the method does not catch the exception locally, it is put into the *EXCEPTIONS* set of that method.

The rules *RTE* and *LE* are shortcuts used to handle the *RunTimeExceptions* and *LinkExceptions* that are thrown by various bytecode instructions.

Bytecode contains an additional `wide` instruction that can be used to modify a following instruction to take *wider* arguments (e.g. a 16-bit local variable number instead of 8-bit). As these instructions produce the same

$$\frac{CALL(M,C,pc,M',R,T,P_1,...,P_n)}{\begin{array}{c} C' := new_context(M,C,pc,M',R,T,P_1,...,P_n) \\ T \subseteq THIS(M',C') \\ P_i \subseteq \bigwedge_{i=1}^{n} PARAM(M',C',i) \\ RETURN(M',C') \subseteq R \\ \bigwedge_{o \in EXCEPTIONS(M',C')} THROW(M,C,pc,o,0) \end{array}}$$

$$\frac{INIT(M,C,pc,c)}{\begin{array}{c} t := \texttt{java.lang.ExceptionInInitializerError} \\ M' := t.\texttt{<init>(java.lang.Throwable)} \\ e := OBJECT(M,pc,t) \\ c\ is\ class \Rightarrow \bigwedge_{o \in EXCEPTIONS(c.\texttt{<clinit>})} \\ \begin{cases} THROW(M,C,pc,o,0) & type(o) \leq \texttt{java.lang.Error} \\ CALL(M,pc,M',e,,,o) & \\ THROW(M,C,pc,e,0) & \text{else} \end{cases} \end{array}}$$

$$\frac{THROW(M,C,pc,o,n)}{\begin{cases} \begin{array}{l} \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq \\ \quad R(M,C,EH_PC(M,n),i) \\ \quad S(M,C,EH_PC(M,n),0) \end{array} & \begin{array}{l} EH_START(M,n) \leq pc \\ \wedge pc < EH_END(M,n) \\ \wedge (EH_TYPE(M,n) = NULL \\ \quad \vee type(o) \leq EH_TYPE(M,n)) \end{array} \\ THROW(M,C,pc,o,n+1) & n < EH_MAX - 1 \\ o \in EXCEPTIONS(M,C) & else \end{cases}}$$

$$\frac{RTE(M,C,pc,t)}{\begin{array}{c} THROW(M,C,pc,OBJECT(M,pc,t),0) \\ type(OBJECT(M,pc,t)) = t \end{array}}$$

$$\frac{LE(M,C,pc,t)}{\begin{array}{c} THROW(M,C,pc,OBJECT(M,pc,t),0) \\ type(OBJECT(M,pc,t)) = t \end{array}}$$

Figure 5.5: Definition of *rules* used in the intra-procedural analysis

$ANALYSIS_DONE$:= 0
while $ANALYSIS_NEEDED \neq ANALYSIS_DONE$
  $\forall (M, C) \in ANALYSIS_NEEDED \setminus ANALYSIS_DONE$:
    $ANALYSE(M, C)$
    $ANALYSIS_DONE$:= $ANALYSIS_DONE \cup (M, C)$

Figure 5.6: Inter-procedural fix-point iteration

constraints as the usual version except for the offsets to the program counter, they were not displayed.

5.3 Inter-Procedural Analysis

For the inter-procedural analysis, it is necessary to analyze all methods reachable by one method. The constraints generated by the intra-procedural analysis have been prepared for an intra-procedural analysis so that constraints from multiple intra-procedural analyzes can be combined into one constraint system. The remaining task is to determine, which methods can be reached, so their constraints must be included. This is generally called call graph construction and there are various algorithms for it (an overview can be found in [51]). Here, the call graph is constructed *on the fly*. In bytecode all method are called by `invoke*` instructions, which generate a CALL rule and this rule puts a pair of method and context into a set called *ANALYSED_NEEDED*. A simple fix-point iteration (figure 5.6) can then be used to include all reachable methods. Existence of this fix-point depends on the function *new_context*, which is used in *CALL*. This function can be used to do an analysis context- or object-sensitive. A simple function $new_context(M, C, pc) = EMPTY$ that will perform an context- or object-insensitive must be used for Snelting/Tip. For this function, the existence of the fix-point is guaranteed, as the number of methods is always limited.

5.4 Whole Program Analysis

The next step is the whole program analysis. Instead of handling individual functions, it analyzes a complete program. The analysis is identical to an inter-procedural analysis starting at the program entry point. Java programs are not monolithic executables with a defined entry method, but a set of classes providing multiple possible entry methods. Usually the user chooses the entry point explicitly by providing the name of class which is searched for a main function. Classes derived from `java.applet.Applet` may additionally be started by an applet viewer. A set of class files may be distributed as a `.jar` file, giving the impression of a monolithic program,

but the jar file just contains the name of the class used as entry point for the `java -jar filename` syntax. The jar file may still contain different entry points and these can still be used by a user with different command-line options.

For this analysis the user must specify one or multiple entry functions, which are analyzed first and the inter-procedural analysis then includes everything reachable from these classes automatically. It would be possible to automatically extract all possible entry points from a given set of classes, but this is not always desirable, as some of these entry points may be debug or test utilities for the containing class, having little to do with the analyzed program.

5.4.1 Stubs

The analysis described so far is sufficient to analyze any program if the complete program code is available in Java bytecode. Unfortunately Java bytecode has the possibility to allow methods to be implemented in other languages (using the `native` keyword). Parts of the library for Sun's JDK are written in C. A hybrid C/Java analysis might be possible, but this will require a lot of work and seems like overkill, because native methods are an exception, not the common case.

However, they still must be taken care of. Simply ignoring native methods will result in wrong analysis results, even for trivial programs.

```
class Sample {
  public String toString() {
    return "a sample";
  }

  public static void main(String args[]) {
    System.out.println(new Sample());
  }
}
```

Inspection of the library reveals, that the field `java.lang.System.out` is initialized inside a native method, which is called within the class constructor of `java.lang.System`. Ignoring this method will leave the field initialized with a dummy objects, which will not call `toString` for an object passed to `println`. Thus the call to `toString` for the parameter object is completely omitted, resulting in an incomplete call graph.

A common technique to handle these kinds of problems are stubs. A stub is a replacement function for a native method. It is written in Java and will be analyzed instead of the native method. It must behave identically as far as the analysis is concerned: it will access the same members, call the

same methods, throw and catch the same exceptions. As an example, the stub for `java.lang.System.arraycopy` is given:

```
public static void
arraycopy(Object src, int src_position,
          Object dst, int dst_position,
          int length)
{
  ((Object[])dst)[0]=((Object[])src)[0];
}
```

For the analysis, most important is the data flow from the content of *src* to the content of *dst*. As array indexes are ignored by the analysis anyway, any index will be sufficient, there is no reason to build a full semantically identically loop. The three exceptions which are specified for the method (`IndexOutOfBoundsException`, `ArrayStoreException`, `NullPointerException`) are automatically included as they may be thrown also by the bytecode instructions for array access. The stubs throws an additional method `CastClassException`, but this is fine for a conservative approximation.

The majority of the stubs are simple and easy, however they sometimes get complicated. They can be created from the methods specification or the implementation. For the Snelting/Tip analysis it is important, that those stubs contain all member accesses and assignment that may happen within the original method.

5.4.2 A Conservative Approximation of Unanalyzed Code

Sometimes it may not be possible to provide a stub for a native method. In these cases a conservative approximation requires to assume that the unknown code will do "everything possible". This generally worsens analysis results significantly, so it is not suitable as a general replacement for writing stubs.

"Everything possible" includes calling of methods, reading and writing of fields or array contents and throwing of arbitrary exceptions. As a worst case, it must be assumed that all unanalyzed methods are communicating with each other. A reference passed as a parameter to one methods may appear as result of another method. Native code can also create new objects. Depending on the signature of the unanalyzed method, this has huge impact. If the return type of unanalyzed methods is `java.lang.Object`, it must be assumed, that the methods may return an instance of every class visible to the analysis. As this might result in loading of hundreds of classes even for small programs is it unpractical. The best solution is to avoid unanalyzed methods and provide stubs. An alternative is to use an heuristic approach

$$\frac{o \in \mathit{UNANALYZED}}{\bigwedge_t^{t \leq type(o)} o \in \mathit{UNANALYZED}_t}$$

$$\frac{o \in \mathit{UNANALYZED}_t}{o \in \mathit{UNANALYZED}}$$

$$\frac{o \in \mathit{UNANALYZED}}{\begin{array}{l}\bigwedge_f f \text{ is a non static field in } type(o) \\ \qquad \Rightarrow \mathit{UNANALYZED}_{type(f)} = \mathit{FIELD}(o,f)\end{array}}$$

$$\frac{o \in \mathit{UNANALYZED}}{\exists \mathit{CONTENT}(o) \mathit{CONTENT}(o) = \mathit{UNANALYZED}}$$

$$\frac{o \in \mathit{UNANALYZED}}{\begin{array}{l}\bigwedge_m m \text{ is a method in } type(o) \Rightarrow \\ \quad o \in THIS(m, EMPTY) \\ \quad \bigwedge_{i=1..PARAMS(m)} \mathit{UNANALYZED}_{TPARAM(m)} \subseteq \\ \qquad\qquad\qquad PARAM(M, EMPTY) \\ \quad RETURN(m, EMPTY) \subseteq \mathit{UNANALYZED} \\ \quad EXCEPTIONS(M, EMPTY) \subseteq \mathit{UNANALYZED}\end{array}}$$

Figure 5.7: Conservative approximation of unanalyzed code

to reduce the number of possible created objects and risk an invalid analysis result.

Figure 5.7 shows the rules for handling with unanalyzed code. There is a general set *UNANALYZED* which contains all objects that may leave the analyzed code. Then there are type-specific sets $\mathit{UNANALYZED}_t$ containing only a subset of *UNANALYZED* that is assignment compatible to t. For all objects of *UNANALYZED*, all their fields and content for arrays, are considered part of *UNANALYZED*. Additionally, all of their methods can be called with any objects from *UNANALYZED* as parameter.

5.4.3 Smart Stubs

Sometimes it is not possible to write a stub for a native method in Java. A simple example might be a functions just made native to bypass access restrictions[3] or static typing. Additionally, the whole functionality of reflection falls into this category. Instead of creating stubs for those methods, special constraints can be added to the constraint system. The stubs example from above can be expressed as:

[3] e.g. `sun.awt.SunToolkit.getPrivateKey`

$$\frac{\texttt{getfield } f \text{ is in } M \text{ at } pc}{\bigwedge_{x \in getrow(S(M,C,pc,0))} (x, f) \in StaticMemberAccess(\mathcal{P})}$$

$$\frac{\texttt{putfield } f \text{ is in } M \text{ at } pc}{\bigwedge_{x \in getrow(S(M,C,pc,1))} (x, f) \in StaticMemberAccess(\mathcal{P})}$$

$$\frac{\texttt{invokeinterface } m \text{ is in } M \text{ at } pc}{\bigwedge_{x \in getrow(S(M,C,pc,PARAMS(M)))} (x, m) \in DynamicMemberAccess(\mathcal{P})}$$

$$\frac{\texttt{invokespecial } m \text{ is in } M \text{ at } pc}{\bigwedge_{x \in getrow(S(M,C,pc,PARAMS(M)))} (x, m) \in StaticMemberAccess(\mathcal{P})}$$

$$\frac{\texttt{invokevirtual } m \text{ is in } M \text{ at } pc}{\bigwedge_{x \in getrow(S(M,C,pc,PARAMS(M)))} (x, m) \in DynamicMemberAccess(\mathcal{P})}$$

Figure 5.8: Constraints for collecting member accesses

$$\frac{CALL(M, \text{pc}, \texttt{java.lang.System.arraycopy}, R, T, P_1, P_2, P_3, P_4, P_5)}{\begin{array}{c} \bigwedge_{s \in P_1} \bigwedge_{d \in P_3} \bigwedge_{o \in CONTENT(s)} type(s) \leq content(type(o)) \Rightarrow o \in CONTENT(d) \\ RTE(M, pc, \texttt{IndexOutOfBoundsException}) \\ RTE(M, pc, \texttt{ArrayStoreException}) \\ RTE(M, pc, \texttt{NullPointerException}) \end{array}}$$

As this version works directly on the sets used as parameters it is more precise than the Java stub version, which will pass the content of every source array to every destination array ignoring whether a specific pair of arrays will be used together in one call or not. The deficit of the Java version can be overcome by analyzing the stub multiple times in different calling contexts.

5.5 Snelting/Tip for this Analysis

The analysis presented so far creates only the points-to sets and assignments, but does not collect member accesses. This can be simply done with the rules in figure 5.8. These five rules cover all member accessing operations, get the objects used as `this`-pointers from the stack and use the `getrow` function to translate the stack content into table rows. Depending on the kind of binding used, the results are put into the corresponding member access set.

For non-member accesses table entries will be created by the rules given in chapter 3.

5.6 Reflection

Reflection refers to a part of the Java API, that enables a user to create an object of a type that is calculated at run-time. Types are represented at run-time as instances of `java.lang.Class`, which can be obtained from another object or by a class name as a string. Reflection is a serious problem for the conservative approximation of a static analysis, as in general every class can be instantiated which results in huge call graphs and analysis results even for small programs.

The usual approach is to force the user to specify which classes can be created at each call site to reflection [49, 37] and therefore basically ignore reflection. But there are at least some things that can and should be done automatically: The `java.lang.Class` object used to create a new instance of an arbitrary class, must be created somewhere in the program. If it is obtained from an other object, the type it represents is known to be the type of that object.

If it is obtained from a string, there is a possibility, that this string is a constant and then the type is known, too. A constant string to create this object is more often used, than one would think, as the Java compiler translates the `.class` operator into calls to reflection. If a Java program contains

```
...
Class c=A.class;
...
```

it is translated into something like[4]:

```
...
Class c=Class.forName("A");
...
```

These strings are constants in Java bytecode. Only if the content of the string is not known, a fall back to user-specified types or a worst case assumption must be made.

Rules for this system can be seen in figure 5.9. Without any further addition, the *OBJECTCLASS* objects will only be created for classes whose type is queried by `getClass` at run-time. This is not a conservative approximation and relies on the user if other classes will be instantiated within the program. A real conservative approximation would require to creation the *OBJECTCLASS* objects for any known class, which results in an unacceptable increase in analysis cost.

The implementation contains another useful heuristic, which is not formalized here: If an object created by `newInstance` reaches a type-cast to

[4]The real transformation is more complicated due to exception handling and caching

$$\frac{CALL(M, \text{pc}, \texttt{java.lang.Class.getClass}, R, T, P_1)}{\begin{array}{c} \bigwedge_{o \in P_1} OBJECTCLASS(type(o)) \in R \\ \bigwedge_{o \in P_1} CLASS(OBJECTCLASS(type(o))) = type(o) \\ \bigwedge_{o \in P_1} INIT(M, pc, type(o)) \\ \bigwedge_{o \in P_1} OBJECTCLASS(type(o)) \in UNANALYZED \\ RTE(M, pc, \texttt{java.lang.LinkageError}) \\ RTE(M, pc, \texttt{java.lang.ExceptionInInitializer}) \\ RTE(M, pc, \texttt{java.lang.ClassNotFoundException}) \end{array}}$$

$$\frac{CALL(M, \text{pc}, \texttt{java.lang.Class.newInstance}, R, T)}{\begin{array}{c} \bigwedge_{o \in T} OBJECT(M, pc, CLASS(o)) \in R \\ \bigwedge_{o \in T} type(OBJECT(M, pc, CLASS(o))) = CLASS(o) \\ RTE(M, pc, \texttt{java.lang.InstantiationException}) \\ RTE(M, pc, \texttt{java.lang.IllegalAccessException}) \end{array}}$$

$$\frac{CALL(M, \text{pc}, \texttt{java.lang.Class.forName}, R, T, P_1)}{\begin{array}{c} \bigwedge_{o \in P_1} \begin{cases} OBJECTCLASS(VALUE(o)) \in R & \exists VALUE(o) \\ UNANALYZED_{\texttt{java.lang.Class}} \subseteq R & \text{else} \end{cases} \\ RTE(M, pc, \texttt{java.lang.LinkageError}) \\ RTE(M, pc, \texttt{java.lang.ExceptionInInitializer}) \\ RTE(M, pc, \texttt{java.lang.ClassNotFoundException}) \end{array}}$$

Figure 5.9: Rules for approximating reflection

a type t somewhere in the program, *OBJECTCLASS* objects for all user-defined classes t' with $t' \leq t$ are created. This is based on the observation, that objects created with reflection are rarely used without being down casted, as without a cast, only the methods of `java.lang.Object` are available. The limitation to user-defined classes is arbitrary and prevents that a cast to a library type (e.g. `java.lang.Comparable`) may result in the creation of far too may objects.

The system presented here only handles a very small amount of the whole reflection API. In order to fully support reflection with regards to Snelting/Tip, the whole API must be implemented, as there are also reflection methods to perform dynamic type-checks for that semantics must be preserved. Access to other constructors, methods and fields can be handled in a similar way to the approach to classes here: for all of them special objects that are tied to the corresponding members must be created. As methods and constructors use arrays to pass information, a more detailed analysis which analyzes array content with higher precision might be useful.

All of these features are currently not contained in the analysis, as they are rarely used in most programs, whereas the simple features of reflection seem to be very commonly used.

5.7 Contexts

The analysis is prepared for the handling of context- and object-sensitivity as described in chapter 4. Details about these techniques are not embedded into the analysis, instead all sets contain a variable C in their name, which makes it easy to analyze a method multiple times with the intra-procedural analysis by supplying different a C every time. As the increased precision of both, context- and object-sensitivity is based on having multiple points-to sets for the same pointer, special knowledge about the nature of the context must not be contained in the intra-procedural analysis, but in the `new_context` function. This function has to decide what context to use for the target of each method call within the program.

Other forms of contexts may also be possible. In a Java program, a possible context would be the thread a method is executed in (although this is similar to object-sensitivity) and then utilized by a smart stub for the method `java.lang.Thread.currentThread` (which returns the object for the current thread).

As context- and object-sensitivity proved incompatible with the Snelting/Tip analysis, similar results seemed probable for other forms of contexts or sensitivity too and this area was not researched in depths.

5.8 Differences to a Real Java Virtual Machine

Although a lot of care has been taken to make the analysis as similar as possible to a real virtual machine, some differences remain. In JDK 1.5, the JVM loads 292 classes before loading the main class specified on the command line. The analysis presented here will not do this, but only load classes that are required for the execution of the program. Loading of these classes is not mandated by the specification of the virtual machine[24], so despite these difference, this analysis can be claimed to be correct. E.g. a real JVM will initialize `java.lang.System` before execution of the `main` method, whereas the analysis will initialize it only if `System` is really used inside the program.

5.9 Implementation

The actual implementation does not use an explicit constraint solver. The sets are represented as nodes in a graph and edges indicate subset relations between sets. All sets for native types are left out from the beginning to

save space. Sets with only one relation to a sub- or superset use a shared representation of the sets content to avoid unnecessary duplication. This is especially efficient for sets modeling local variables, as these sets are usually changed by only a few instructions of each method. All conditional constraints are attached to the nodes of the graph. As soon as the content of the set is changed, the constraints are rechecked. Source-code level local variables are excluded from meta-sets per default in order to have identical results for bytecode with or without debugging information. In addition, this has a similar result to transforming the bytecode of a method into single static assignment form, which enhances analysis results. This analysis is fast in practice and the code analysis alone (including detecting member accesses, but without propagating points-to information) took less than a second on a machine four years ago[5]. If points-to propagation is included, the time for analyzing a single method depends almost completely on the time of the propagation.

[5] 800 MHz Athlon, 256 MB RAM, Redhat Linux 7.1

Chapter 6

Dynamic Snelting/Tip

Dynamic program analysis is an alternative to static program analysis. Instead of making statements about certain properties of a program that are true for every possible program run, a dynamic analysis observes these properties while running the program. This can be helpful if a static analysis is either to expensive or too imprecise and the analysis results are not required to hold for every program run.

For Snelting/Tip a dynamic analysis provides two advantages: A dynamic analysis avoids any scalability issues with the points-to analysis and it solves all problems concerning the use of reflection, even without manual input by the user.

6.1 Creating the Table at Run-Time

The main limitation of the static analysis is the memory consumption of the points-to graph. To avoid this, the dynamic analysis omits pointers completely and focuses only on member access through objects. This is a deliberate limitation, there are other dynamic analysis which take pointers and points-to into account[20].

The base for a dynamic analysis is a number of program runs:

Definition 21 Let $\mathcal{R}$ be an arbitrary number of program executions for a program $\mathcal{P}$. Then $s \in \mathcal{R}$ is true iff s is a statement in $\mathcal{P}$ and s is executed during $\mathcal{R}$. For all variables $x_1, \ldots, x_n$ used in s, $\mathcal{R}$ will provide sets of objects $\mathcal{R}(x_1), \ldots, \mathcal{R}(x_n)$ which contain all objects that the variables may reference in $\mathcal{R}$.

In order to keep results from dynamic and static analysis as interchangeable as possible, the same notation will be used for both.

Definition 22 The sets of pointers and objects for a dynamically analyzed program $\mathcal{P}$ are defined as follows:

$$\begin{aligned}
&\mathit{Pointers}(\mathcal{P}) := \{\} \\
&\mathit{Objects}(\mathcal{P}) := \{s | \texttt{new C} \text{ in } \mathcal{P} \text{ at } s \text{ and is executed in } \mathcal{R} \\
&\qquad\qquad\qquad \mathit{type}(s) = C, C \text{ is a class in } \mathcal{P}\}
\end{aligned}$$

As pointers are omitted from the analysis, member declarations become redundant and only method definitions are used.

Definition 23 Member declarations and definitions for a dynamically analyzed program $\mathcal{P}$ are defined as follows (constructors are not considered methods):

$$\begin{aligned}
&\mathit{MemberDcls}(\mathcal{P}) := \{\} \\
&\mathit{MemberDefs}(\mathcal{P}) := \{\mathit{def}(C.m) | m \text{ is not an abstract method in } C, \\
&\qquad\qquad\qquad C \text{ is a class in } \mathcal{P}\}
\end{aligned}$$

Compared to the static analysis, table entries for member access can be generated very easily:

Definition 24 The table entries for a dynamically analyzed program $\mathcal{P}$ due to member access operations are defined as follows:

$$\frac{p.m \text{ occurs in } \mathcal{P} \text{ and is reached in } \mathcal{R} \text{ with } o \in \mathcal{R}(p)}{\begin{array}{c} X := \mathit{LookupDefinition}(\mathit{type}(o), m) \\ (o, \mathit{def}(X.m)) \in T \end{array}}$$

The reduced complexity for accesses is caused by the omission of pointers. Removing pointers also removes static types, making the whole lookup procedure easier.

The removal of pointers also removes assignments, because the left hand side of an assignments in Java can only be a pointer in Java.

Definition 25 The set of assignments for a program $\mathcal{P}$ is defined as follows:

$$\mathit{Assignments}(\mathcal{P}) := \{\}$$

The rest of the analysis remains theoretically unchanged, but in practice becomes easier, too. The omission of assignments and the implications generated for them removes the requirement to do a fix-point iteration for creation of the final table. But as this iteration is not very expensive, this is only a minor benefit.

Please notice that the lattice created from a table without pointers is different from the lattice created from a table where pointers are included for propagation, but omitted for lattice generation.

The dynamic analysis does not make any assumptions about the number of program runs used to extract information for the table. A single run is sufficient, but results from multiple runs can be easily integrated. Of course,

```
1   class A {
2      void f() {}
3      void g() {}
4   }
5   ...
6   boolean b=...
7   A a;
8   if(b)
9      a=new A(); // A1
10  else
11     a=new A(); // A2
12  if(b)
13     a.f();
14  else
15     a.g();
```

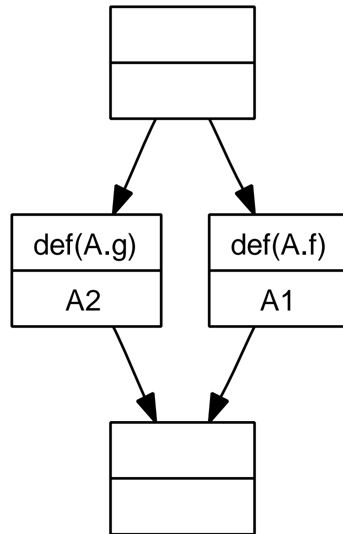


Figure 6.1: Example program for limitations of the dynamic analysis

if multiple runs are combined and provide a better coverage of the analyzed program, the resulting class hierarchy will be more general and more similar to the result of the static analysis. Usually a test suite for the analyzed program should be used, as in general test suites are built to cover as much code as possible of the program. A useful minimum should at least cover all `new` statements within the program.

6.2 Limitations

It should be quite obvious that the omission of pointers and member declarations has an effect on the created class hierarchy. Figure 6.1 shows a small example program and the concept lattice resulting from the dynamic analysis[1] In this program, the original class A will be horizontally split into a class containing only member f and another class containing only g, as for all program runs, $A1$ will only access f and $A2$ will only access g. At least a common supertype for $A1$ and $A2$ is required in order to provide a new type for the pointer a. A static analysis is usually not able to find out the access pattern and creates only one type for $A1$ and $A2$.

It is important to notice that this effect is not exclusively caused by the omission of pointers and declarations. The result from a (hypothetical) dynamic analysis including these can be seen in figure 6.2. Here a type for a exists, but the types for $A1$ and $A2$ are abstract and thus the whole class hierarchy not realizable.

[1] assuming multiple program runs where b is at least once `true` and once `false`.

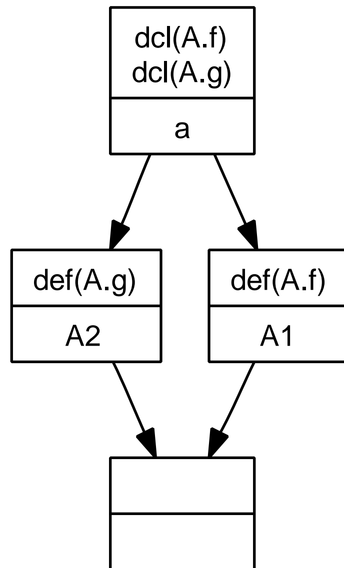


Figure 6.2: Class hierarchy of figure 6.1 including pointers

It is interesting to see that the effects encountered here produce the same kind of "defects" in class hierarchies that some enhancements for the points-to analysis already did. Of course the flow-sensitivity that broke the static points-to analysis for Snelting/Tip, is automatically present in a dynamic analysis and the same is true for context- and object- sensitivity. Unfortunately, the dynamic analysis has far fewer ways, to "fix" these problems.

Still, the dynamic variant can be useful to find out, how objects are used within a program and get practical suggestions for modifications to the class hierarchy. On a JVM with all static type-checking removed, the class hierarchy would still be valid and preserve the semantics of the original program. This indicates that the dynamic analysis can be used for application of Snelting/Tip to languages which do not suffer from strong static type systems, e.g. Smalltalk.

6.3 Implementation

The implementation was done by instrumenting the bytecode interpreter Java virtual machine *kaffe*[2]. The reason for choosing this JVM was its plain structure and easy to understand code. In fact, most of the instrumentation for *kaffe* was already done, before the compilation of Sun's JVM (which was started in parallel) even finished. The instrumentation uses a compact representation in memory and causes very low run-time overhead (less than

[2] `www.kaffe.org`

10 percent). The result is written into a single file, where basically each line represents a table entry. Multiple of these files can be merged by a small tool into the same output format produced by the static analysis. This kind of instrumentation should be easily applicable to other virtual machines as well.

Chapter 7

Semantic Preserving Refactoring

The original article by Snelting and Tip[43] already gives ideas for a tool that takes the result of the analysis and allows interactive restructuring of the class hierarchy. The target audience for such a tool is an engineer who is familiar with the code, but is not required to have in-depth knowledge about the Snelting/Tip analysis.

Besides providing the editing capability, such a tools must still check if manual changes to the class hierarchy will affect the program semantics. Except for trivial programs, even someone familiar with the source code of an analyzed program, might have problems to decide, whether e.g. the moving of a method up in the lattice will affect the semantics of the program. So it is one task of the tool to check the correctness of manually applied transformations automatically and give the user appropriate feedback.

As knowledge of all details of the Snelting/Tip analysis is a very high requirement for the user of such a tool, it seems more appropriate to create a user interface, which hides the theoretical background as much as possible. The concept lattice can be displayed in a way similar to the display of class hierarchies in UML-tools. The bottom element can be removed and dead members displayed in a separate location.

For the user interface this difference may just seem like cosmetics, but it has huge implications on what such a tools does behind the scenes. Such a tool can be done in two ways:

- The tool uses the original data structures from the Snelting/Tip analysis and every modification to the class hierarchy is translated into a manipulation of these data structures. After manipulation, a new class hierarchy can be calculated which incorporates the changes wanted by the user.

- The tool uses alternate data structures which are initialized with the

results of the analysis. These data structures allow checking if a modification of the class hierarchy affects program semantics and are directly manipulated by applying such modifications.

At first glance, keeping the data structures from the analysis seems the easier way, as no second set of data structures is needed. But in practice this approach provides some problems. If e.g. the user moves a member up in the hierarchy, new table entries will be generated, the iteration must be re-run and then a new concept lattice must be calculated. This calculation is too expensive in practice (see 10.5 for running times of the lattice construction) and there is no incremental algorithm which would make use of the fact, that only a minor change was done to the table. Even if creating the concept lattice would be fast, a delta between the old lattice and new lattice is needed to update the graphical display - otherwise it would be required to create a new graphical layout for the new lattice, which might result in bigger changes to the layout than the user expected.

After some very unsatisfying experiments with such an editor (it was basically unusable as calculating the lattice increased response time to user actions dramatically), the second approach was chosen. Modifications by the user are handled in a similar way to transactions in database: before a commit it is checked if the modified hierarchy will modify the program semantic compared to the semantic of the initial class hierarchy created by the concept analysis. If there is a difference, the transaction is rolled back, otherwise the commit can be applied.

While this provides a big improvement in terms of response time and therefor usability, it also provides some drawbacks. (1) Some operations which are easy to implement on the original table (e.g. adding a new assignment) are hard to do with this model. They can be done manually, but hardly automatically. (2) Potential implications between table rows, that would be automatically created after a manipulation of the table, must be added manually. If such an implication is required for the preservation of the program semantic, it must be added manually by the user. This is however, not a big problem in practice.

7.1 Class Hierarchy for a Concept Lattice

The concept lattice can be transformed into a directed acyclic graph. As explained earlier, this is a more appropriate representation for a class hierarchy, as it can directly be manipulated by the user and provides more freedom (every concept lattice can be transformed into a graph, but not vice versa).

A class hierarchy $C = (G, Attrs, Objs)$ based on the concept lattice $\mathcal{L}(T)$ is defined as:

$$G = (N, E)$$
$$N = \{n_c | c \in \mathcal{L}(T) \wedge (c \neq \bot \vee \exists o : \gamma(o) = c)\}$$
$$E = \{(n_{c_1}, n_{c_2}) | n_{c_1} \in N, n_{c_2} \in N, c_2 < c_1 \wedge \nexists c_3 : c_2 < c_3 < c_1\}$$

This graph omits the bottom node if it is not labeled with any object (which is the usual case), because it will not be a useful class in general.

Nodes of the graph are labeled with the attributes and objects from the concept lattice as follows:

$$Attrs(n_c) = \{a | \mu(a) = c\}$$
$$Objs(n_c) = \{o | \gamma(o) = c\}$$

Some additional helper definitions are used:

$$n \leq n' \Leftrightarrow n = n' \vee \bigvee n_1, n_2, ..., n_i : (n', n_1) \in E \wedge (n_1, n_2) \in E \wedge ... \wedge (n_i, n) \in E$$
$$Class(a) = \{n | a \in Attrs(n)\}$$
$$Type(o) = \{n | o \in Objs(n)\}$$
$$Sigs(n) = \bigcup_{n' \in N}^{n \leq n'} \{m | \exists X : def(X.m) \in Attrs(n') \vee dcl(X.m) \in Attrs(n')\}$$
$$Checks(n) = \bigcup_{n' \in N}^{n \leq n'} \{(T, M, pc) | check(T, M, pc, r) \in Attrs(n')\}$$

7.2 The Lookup Algorithm

The class hierarchy derived from the concept lattices will utilize multiple inheritance for almost all non-trivial programs, although the original Java programs did not use multiple inheritance. This is a result of the minimality of the concept lattice. While this may be useful to gain insight on the object and class usage, it is not appropriate in a restructuring proposal for a Java program. The problem of removing multiple inheritance will be tackled later (see 8.4), another problem is relevant first.

In a class hierarchy with multiple inheritance, it is sometimes not obvious, which method implementation will be chosen for a class if different implementations are visible in different superclasses. Snelting/Tip suggested using the lookup algorithm that is used by C++ [1]. A version of that algorithm that works on our notion of graphs and member declaration and definition, is shown in figure 7.1. The function *lookup* returns all candidates matching signature m, starting from node n.

Of course this algorithm is an obvious candidate for using it for Java too. But closer inspection reveals, that this algorithm is not compatible with Java's semantics for interfaces. Interfaces are similar to C++ classes

```
hides(a,b) :=
   if Type(a) = Type(b) then
      return a ≡ def(C.m)  ∧  b ≡ dcl(D.m)
   else
      return Type(a) ≤ Type(b)
   fi

lookup(n,m) :=
   cand:={a|n ≤ Type(a)  ∧
              (a ≡ def(C.m) ∨ a ≡ dcl(C.m)) }

   return {a ∈ cand ∧ ∄ b ∈ cand : hides(b,a) }
```

Figure 7.1: Lookup algorithm for C++

containing only abstract method declarations and handled that way in the analysis. The relevant difference in this context is their influence on method lookup in the context of multiple inheritance. During lookup, an abstract method in C++ is treated equally to a method definition. In Java, a method declared in an interface will not override a method definition from a superclass and only add a method declaration if a method with a matching signature is not already present.

Figure 7.2 shows a small example program. The class B inherits an implementation of the method f from its superclass A and a declaration from the interface I. In the corresponding C++ program (figure 7.3) B is an abstract class and cannot be instantiated (the program cannot be compiled).

It is easy to adjust the algorithm for Java. Figure 7.4 shows a revised *hides*. Here, possible overwriting of a definition by a declaration from an interface is handled as a special case. This addition will not affect the result of *lookup*, if the result was unambiguous with the original algorithm, but change some cases from ambiguous to unambiguous (like the given example).

7.3 Preserving Semantics for Modifications

The class hierarchy represented by the graph created initially from the concept lattice. If the graph is then modified, it must be explicitly checked, whether the modification preserves semantics. Modification in this context means every possible change to the content of C.

The usual way of modifying a class hierarchy would be to apply a number of small changes one after another, creating a series of hierarchies $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow ... \rightarrow C_n$. It is possible to check if semantics get broken for each

```
 1  class A {
 2     public void f() {}
 3  }
 4  interface I {
 5     void f();
 6  }
 7  class B extends A implements I {
 8  }
 9  class Test {
10     static A a;
11     static I i;
12
13     public static void main(String args[]) {
14        a=new A();
15        i=new B();
16
17        a.f();
18        i.f();
19     }
20  }
```

Figure 7.2: Multiple inheritance with interfaces

```
 1  class A {
 2  public:
 3     virtual void f() {}
 4  };
 5  class I {
 6  public:
 7     virtual void f() = 0;
 8  };
 9  class B : public A, public I {
10  };
11
12  static A *a;
13  static I *i;
14
15  int main(int,char **) {
16    a=new A();
17    i=new B(); // compile error
18
19    a->f();
20    i->f();
21  }
```

Figure 7.3: Multiple inheritance with abstract classes

```
hides(a, b) :=
   if a ≡ def(C.m) ∧ b ≡ dcl(D.m) ∧ D is an interface
      return true
   else if b ≡ def(C.m) ∧ a ≡ dcl(D.m) ∧ D is an interface
      return false
   else if Type(a) = Type(b) then
      return a ≡ def(C.m) ∧ b ≡ dcl(D.m)
   else
      return Type(a) ≤ Type(b)
   fi
```

Figure 7.4: Lookup algorithm for Java

step $C_i \to C_{i+1}$, but this is too strict, as some modifications may not be revertible as $C_{i+1} \to C_i$ might not be a semantics preserving transformation. Instead, each C_i is checked against C_1, making it possible to revert a previous transformation.

In order for a modified class hierarchy C' not to break the semantics of an original class hierarchy C, a number of conditions must be fulfilled. In the following, G', N', E', $Attrs'$, $Objs'$, $Class'$, $Type'$, $Sigs'$, $Checks'$, $lookup'$ and $\leq'$ refer to the modified hierarchy instead of the original hierarchy.

No pointer or object is removed:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)} \exists c' \in N' : o \in Objs'(c') \tag{7.1}$$

No pointer looses a member declaration:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)}^{o \text{ is a pointer}} \bigwedge_{m \in Sigs(Type(o))} lookup'(Type'(o), m) \neq \{\} \tag{7.2}$$

No object or pointer looses a member definition:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)} \bigwedge_{x \equiv def(X.m)} Type(o) \leq Class(x) \Rightarrow Type'(o) \leq' Class'(x) \tag{7.3}$$

Assignments must stay type-correct:

$$\bigwedge_{(p,q) \in Assignments(\mathcal{P})} Type'(q) \leq' Type'(p) \tag{7.4}$$

No class for an object becomes abstract:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)}^{o \text{ is an object}} \bigwedge_{m \in Sigs(Type(o'))} \nexists C : dcl(C.m) \in lookup'(Type'(o), m) \tag{7.5}$$

Method lookup for all objects is unchanged:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)}^{o \text{ is an object}} \bigwedge_{m \in Sigs(Type(o))} \exists C : C.m \text{ is a non-static method} \wedge \tag{7.6}$$
$$lookup(Type(o), m) = \{def(C.m)\} \Rightarrow lookup'(Type'(o), m) = \{def(C.m)\}$$

Successful type-checks stay successful:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)} \bigwedge_{T,M,pc} Type(o) \leq Class(check(T, M, pc, true)) \Rightarrow \tag{7.7}$$
$$Type'(o) \leq' Class'(check(T, M, pc, true))$$

Unsuccessful type-checks do not become successful:

$$\bigwedge_{c \in N} \bigwedge_{o \in Objs(c)} \bigwedge_{T,M,pc} Type(o) \leq Class(check(T, M, pc, false)) \Rightarrow \tag{7.8}$$
$$Type'(o) \nleq' Class'(check(T, M, pc, true))$$

A $\top$ element exists in the class hierarchy:

$$\bigvee t \in N' \bigwedge_{c \in N'} c \leq' t \tag{7.9}$$

The first three conditions 7.1, 7.2 and 7.3 ensure the completeness of all objects and pointers. It terms of the table T, they guarantee that no row has been removed from T and no entry (relevant) $(o, a) \in T$ has been removed.

The condition 7.4, which checks if all assignments stay type-correct, refers back to the set of assignments from the analysis, as this set is smaller and thus less restrictive, than an assignments set inferred from existing subtype-relations in C would be.

The must important conditions for preserving semantics are 7.6, 7.7 and 7.8, as they guarantee that the behavior of dynamic binding and type-checks remain unchanged after the modification. 7.6 plays the same role as the hiding implications in the table. While the implications forced a specific *def* to be the lowest reachable for a specific object, this condition only checks if this was changed. Conditions 7.7 and 7.8 are asymmetrical,

as both check only against the *true* attribute. Alternately, the check in 7.8 can be done against the *false* attribute, but then an additional check $\mathit{Type}'(\mathit{check}'(T, M, pc, \mathit{false})) \nleq \mathit{Type}'(\mathit{check}'(T, M, pc, \mathit{true}))$ is required.

Condition 7.5 ensures that every class can be instantiated, if an object of that class exists.

The final condition 7.9 is required as an artifact of the implementation. As Java programs always have `java.lang.Object` on top of every class hierarchy, adding this condition is no relevant restriction of the class hierarchy.

The set of conditions looks very restrictive at first glance, but it is not. All objects and pointers can be extended with additional member declarations or definitions. In fact, pointer may be changed to use a different declaration (or even a definition) than they did originally, which was not possible using the original table and concept lattice.

A straight forward implementation of a check for these conditions may result in a very expensive implementation. The highest cost is caused by the *lookup* function, which has to traverse the graph to a certain degree. In many Java programs, calls to the method `toString` and `hashCode` will be found and many classes don't overwrite the default implementation in `java.lang.Object`. In this case, every call to *lookup* has to traverse the class hierarchy from the class of the object to the top of the hierarchy where the implementations can be found.

As this is a very relevant problem in practice, the next section will contain special data structures and algorithms to implement the check these conditions in an efficient way.

7.4 Efficient Check for Broken Semantics

The required conditions to ensure, that a modification does not break the program semantic, are quiet big and must be carefully implemented in order to be efficient. Two observations helped in speeding up this process:

(1) The semantics of a modified hierarchy C_i is always compared to the original hierarchy C. As C stays unmodified the whole time, expensive calls to functions like *lookup* can be cached.

(2) Many modifications from C_i to C_{i+1} will be local changes and do not affect all classes. A change to a class n only affects another class m if $m \leq n$, so only these classes have to be rechecked.

7.4.1 Caching Semantic Information

As a cache for all semantic information of a class n, a structure $V(n) = (S_n, D_n, T_n)$ is used, where S_n is a set of class members that must be available in n for static binding, D_n is a map which stores the result of $\mathit{lookup}(m, n)$ for each m, so $D_n(m) = \mathit{lookup}(m, n)$. Similarly, T_n contains the type-check attributes $\mathit{check}(T, M, pc, r)$ for each (T, M, pc).

Each $V(n)$ does not need to be constructed individually, instead it can be constructed from $Attrs(n)$ and $\{V(m)|(m,n) \in E\}$. This allows total elimination of the usage of *lookup*, replacing the searching algorithm by a constructive variant. This algorithm *create_cache* is shown in figure 7.5.

7.4.2 Creating Incremental Caches

If a modification is made to a class hierarchy, obviously only the classes modified and the classes below these can be affected by those changes. Classes above the modified classes remain unchanged and so does their cache $V(n)$. This can be used to construct the cache for a modified hierarchy faster, if the set of modified nodes is known.

The algorithm for incremental construction assumes the cache of the hierarchy before modification is still present in V', as well as the original set of attributes for a node as $Attrs'(n)$. It takes a number of start nodes and begins construction of new caches at these start nodes. Obviously, a node was modified, if the modification changed its *Objs* or *Attrs* set, but also if an incoming edge has been added or removed. The algorithm processes a node only if all nodes above it have either been processed or were unaffected by the change. Then a new cache is only constructed if the attribute of node changed or the cache for one of its superclasses changed. Subclasses of the node are marked for processing only if the cache for the node is changed[1].

7.4.3 Comparing Two Caches

The practical use of the caches presented is an easy way to determine whether a class $n' \in N'$ from a modified hierarchy C' can be used as a replacement for a class $n \in N$ of the original hierarchy C. Two different versions of comparison are required: for objects and pointers.

For objects it must be checked, whether all static members are still included, the lookup for dynamic bound methods has not changed, the result of type-checks have not changed and the class can still be instantiated. The check for pointers is easier: Only the availability of static members and the existences of at least a declaration for dynamic bound methods must be checked.

The algorithm to compare two complete hierarchies is shown in figure 7.9. With the assumption that an undefined result of *Type* will cause the whole algorithm to fail, this algorithm checks all conditions presented earlier, except for 7.4 and 7.9.

An implementation of the algorithm lead to a tremendous speed up. For medium sized class hierarchies (about 1000 classes), the algorithm to

[1] Depending on the cost of various functions involved and the size of the graph, it might be faster to omit checking if the cache for a node has really changed and rebuild all caches for nodes below it.

`type_hides`(a, b) `:=`
  `return` $a \equiv \mathit{check}(T, M, pc, \mathit{true})$

`construct`(n, pred) `:=`
  $S_n := \bigcup_{p \in \mathit{pred}} S_p \cup \{a | a \in \mathit{Attrs}(n) \wedge a \equiv \mathit{def}(C.m)\}$

  $\mathit{cand} := \{\}$
  $\forall m \in \mathit{Sigs}(m)$
    $\mathit{cand} := \bigcup_{p \in \mathit{pred}} D_p(m) \cup \{a | a \in \mathit{Attrs}(n) \wedge a \equiv \mathit{def}(C.m) \vee a \equiv \mathit{dcl}(C.m)\}$
  $D_n(m) := \{a \in \mathit{cand} \wedge \nexists b \in \mathit{cand} :$ `hides`$(b, a)\}$

  $\mathit{cand} := \{\}$
  $\forall (T, M, pc) \in \mathit{Checks}(m)$
    $\mathit{cand} := \bigcup_{p \in \mathit{pred}} T_p(T, M, pc) \cup \{a | a \in \mathit{Attrs}(n) \wedge a \equiv \mathit{check}(T, M, pc, x)\}$
  $T_n(T, M, pc) := \{a \in \mathit{cand} \wedge \nexists b \in \mathit{cand} :$ `type_hides`$(b, a)\}$

`create_cache() :=`
  append $\top$ to *todo*
  `while` *todo* $\neq$ $\{\}$
    assign n to the first element of *todo*
    remove first element of *todo*

    $\mathit{pred} := \{p | (p, n) \in E\}$

    `if` $\nexists\ p \in \mathit{pred} : \mathit{not}\ \mathit{visisted}(p)$
      `construct`(v, pred)`;`

      $\forall m \in \{m | (n, m) \in E\}$
        append m to *todo*

      $\mathit{visited}(n) := \mathit{true}$

Figure 7.5: Construction algorithm for caches of semantic information

```
create_cache_incremental(modified) :=
  ∀n ∈ modified
    append n to todo
  ∀n ∈ N
    V(n):=V′(n)
  while todo ≠ {}
    assign n to the first element of todo
    remove first element of todo

    if ∄ p ∈ todo : n ≤ p then
      pred := {p|(p, n) ∈ E}

      if Attrs(p) ≠ Attrs′(p) ∨ ∃ p ∈ pred : V(p) ≠ V′(p) then
        construct(v, pred)

        if V(p) ≠ V′(p) then
          ∀m ∈ {m|(n, m) ∈ E}
            append m to todo
        fi
      fi
    else
      append n to todo
    fi
```

Figure 7.6: Incremental construction of semantics information

```
compare_for_object(n,n′) :=
  if not S_n ⊆ S′_n′
    return false
  ∀ m : D_n has entry D_n(m)
    if not D_n(m) = {def(C.m)} ⇒ D′_n′(m) = {def(C.m)}
      return false
  ∀ m : D′_n′ has entry D′_n′(m) ∧ D_n has no entry D_n(m)
    if ∄ C : D′_n′(m) = {def(C.m)}
      return false
  ∀ (T, M, pc) : T_n has entry T_n(T, M, pc)
    if not T_n(T, M, pc) = {check(T, M, pc, true)} ⇒
           T′_n′(T, M, pc) = {check(T, M, pc, true)}
      return false
  return true
```

Figure 7.7: Compatibility test for objects

```
compare_for_pointer(n,n') :=
  if not S_n ⊆ S'_n'
    return false
  if ∃ m : D_n has entry D_n(m) ∧ D'_n' has no entry D'_n'(m)
    return false
  return true
```

Figure 7.8: Compatibility test for pointers

```
compare_hierarchy(modified) :=
∀ n ∈ N : ∃ m : n ≤ m
  if ∃o ∈ Objs(n) : o is object then
    ∀n' ∈ {n'|n' = Type'(o)}
      if not compare_for_object(n, n')
        return false
  fi
  if ∃o ∈ Objs(n) : o is pointer then
    ∀n' ∈ {n'|n' = Type'(o)}
      if not compare_for_object(n, n')
        return false
  fi
```

Figure 7.9: Compatibility test for whole hierarchy

reconstruct library classes (see 8.2), was brought from multiple days running time to a few seconds.

Multiple reasons are responsible for this speed up:

- The use of caches as replacement for *lookup*
- Doing incremental checking only for modified parts of the class hierarchy
- Comparing groups of objects that have the same type instead of individual objects
- The results for the $\leq$ function were additionally cached

As alternative to caching, it was tried to replace $\leq$, which can be implemented with depth-first or breads-first search, with a function of constant complexity by calculating the transitive hull $O(n^3)$ upfront. Practice showed that this initial overhead is too big, especially as the transitive hull must to be rebuilt after each structural change to the graph, often causing high overhead for small changes in the class hierarchy.

Chapter 8

Refactoring Algorithms

This chapter describes methods and algorithms used to manipulate the class hierarchy. The semantic preserving checks presented earlier were generic for all possible modifications of the class hierarchy. In practice, manipulation can be split into the basic actions, that actually modify N, E, *Attrs* or *Objs* and more complex algorithms that are composed of these basic actions.

Almost all algorithms are presented in a slightly simplified form that omits some checks required to deal with the fact that the algorithm is working on a graph that is currently changing. This was done to focus on the nature of the algorithms and omit implementation details.

Not all algorithms automatically preserve semantics. Some of them make modifications and then check if semantics are preserved. Of course, if not, the modifications have to be reverted. In order to keep the algorithm code free of this procedure, a special construct in pseudo code is used:

```
try
    // make some modifications to graph here and check semantics
    // if the end of the block is reached
success
    // execute this code only if the modifications preserve semantics
failure
    // automatically revert the changes made in the try block
    // and execute this code
```

The `success` and `failure` blocks are optional, of course even with a missing `failure` block, the changes are still reverted.

8.1 Basic Manipulation of the Class Hierarchy

The small functions shown in this section provide the basic blocks for all following algorithms. These are the only functions actually modifying the graph. Each functions describes the construction of a new graph C' from

C, based on additional parameters. While the first five algorithms are self-explanatory and trivial, the other deserve a word of explanation. `merge` collapses two classes into one. If these two classes do not have a direct inheritance relation all classes between the two are collapsed as well. All incoming and outgoing edges to the collapsed classes are preserved. `distribute` removes an attribute from a class and places it at all successors. This may make an attribute appear multiple times in the class hierarchy. Currently, this is only used for $check(T, M, pc, false)$ attributes, as duplication of other attributes is problematic and usually unwanted, as it results in code duplication. `shortcut` removes an class without attributes or objects by directly connecting its sub- and superclasses.

`move_attribute`(n, a, n') `:=`
Precondition: $n, n' \in N \;\wedge\; a \in Attrs(n)$
$N' := N$
$E' := E$
$\forall\; m \in\; N \;:\; m \neq n \wedge m \neq n' \;:\; Attrs'(m) := Attrs(m)$
$Attrs'(n) := Attrs(n) \setminus \{a\}$
$Attrs'(n') := Attrs(n') \cup \{a\}$
$Objs' := Objs$

`move_object`(n, o, n') `:=`
Precondition: $n, n' \in N \;\wedge\; o \in Objs(n)$
$N' := N$
$E' := E$
$Attrs' := Attrs$
$\forall m \in N \;:\; m \neq n \wedge m \neq n' \;:\; Objs'(m) := Objs(m)$
$Objs'(n) := Objs(n) \setminus \{a\}$
$Objs'(n') := Objs(n') \cup \{o\}$

`add_edge`(n, m) `:=`
Precondition: $n, m \in N \;\wedge\; (n, m) \notin E$
$N' := N$
$E' := E \cup \{(n, m)\}$
$Attrs' := Attrs$
$Objs' := Objs$

`remove_edge`$(n, m))$ `:=`
Precondition: $n, m \in N \;\wedge\; (n, m) \in E$
$N' := N$
$E' := E \setminus \{(n, m)\}$
$Attrs' := Attrs$
$Objs' := Objs$

`add_node`(n) `:=`

$\texttt{\textit{Precondition:}}\ n \notin N$
$N' \texttt{:=} N \cup \{n\}$
$E' \texttt{:=} E$
$Attrs' \texttt{:=} Attrs$
$Objs' \texttt{:=} Objs$

$\texttt{merge}(n, n')\ \texttt{:=}$
$\texttt{\textit{Precondition:}}\ n, n' \in N \quad \wedge \quad n \neq n' \quad \wedge \quad n' \leq n$
$remove \texttt{:=} \{m | m \leq n \quad \wedge \quad m \neq n \quad \wedge \quad n' \leq m\}$
$N' \texttt{:=} N \setminus remove$
$E' \texttt{:=} E \setminus \{(m, m') | m \in remove \quad \vee \quad m' \in remove\}$
$\cup \{(n, m) | \exists (x, m) \in E \quad : \quad x \in remove \quad \wedge \quad m \notin remove\}$
$\cup \{(m, n) | \exists (m, x) \in E \quad : \quad x \in remove \quad \wedge \quad m \notin remove\}$
$\forall m \in N : m \neq n$
$Attrs'(m) \texttt{:=} Attrs(m)$
$Attrs'(m) \texttt{:=} Attrs(m) \cup \bigcup m \in remove : Attrs(m)$
$\forall m \in N : m \neq n$
$Objs'(m) \texttt{:=} Objs(m)$
$Objs'(m) \texttt{:=} Objs(m) \cup \bigcup m \in remove : Objs(m)$

$\texttt{distribute}(n, a)\ \texttt{:=}$
$\texttt{\textit{Precondition:}}\ n \in N \quad \wedge \quad a \in Attrs(n)$
$N' \texttt{:=} N$
$E' \texttt{:=} E$
$next \texttt{:=} \{m | (n, m) \in E\}$
$\forall \ m \in N \quad : \quad m \neq n \quad \wedge \quad m \notin next$
$Attrs'(m) \texttt{:=} Attrs(m)$
$\forall \ m \in next$
$Attrs'(m) \texttt{:=} Attrs(m) \cup \{a\}$
$Attrs'(n) \texttt{:=} Attrs(n) \setminus \{a\}$
$Objs' \texttt{:=} Objs$

$\texttt{shortcut}(n)\ \texttt{:=}$
$\texttt{\textit{Precondition:}}\ n \in N \quad \wedge \quad Attrs(n) = \{\} \quad \wedge \quad Objs(n) = \{\}$
$N' \texttt{:=} N \setminus \{n\}$
$E' \texttt{:=} E \setminus \{(m, m') | m = n \quad \vee \quad m' = n\}$
$\cup \{(m, m') | (m, n) \in E \quad \wedge \quad (n, m') \in E\}$
$Attrs' \texttt{:=} Attrs$
$Objs' \texttt{:=} Objs$

8.2 Creation of Library Classes

As explained in section 3.8.2, a class hierarchy constructed from the concept lattice will have the members of library classes scattered among different

classes. But in almost all cases the refactored program is supposed to run with the original class hierarchy, so it is required to create classes within the new hierarchy that are equal to the original library classes.

The following algorithm will "collect" all member attributes for a class and place them at the node containing the class attribute. In this process, objects and pointers are merged to the same node.

```
add_above(c, t) :=
  if (c, t) ∉ aboves
    add_node(above_c^t)
    aboves:=aboves ∪ {(c, t)}

create_library_classes() :=
  todo:={c|∃n : all(c) ∈ Attrs(n)}
  done:={}

  while ∃c ∈ todo : ∄t ∈ todo : t ≤ c
    todo:=todo \ {c}

    // Step 1
    made_smaller:={}
    below:={n|n ≤ Class(all(c))}
    ∀b ∈ below:
      ∀a ∈ Attrs(b):
          if a ≡ dcl(X.m)  ∨  a ≡ def(X.m)
              ∨ a ≡ check(X, M, pc, true)
              if c = X
                if b ≠ Class(all(c))
                  move_attribute(b, a, Class(all(c)))
                  made_smaller:=made_smaller ∪ {b}
              else if c ≤ X
                add_above(c, X)
                move_attribute(b, a, above_c^X)
                made_smaller:=made_smaller ∪ {b}
      ∀o ∈ Objs(b):
          if c = type(o)
              above:=above ∪ {b}
          else if c ≤ type(o)
              add_above(c, type(o))
              move_object(b, o, above_c^type(o))
              made_smaller:=made_smaller ∪ {b}
          else if type(o) ∈ done  ∧  b ∉ cleaned
              if b ≠ Class(all(c))
                  move_object(b, o, Class(all(c)))
```

$$made_smaller := made_smaller \cup \{b\}$$

```
// Step 2
while ∃ m ∈ below \ above : m ∈ N  ∧  m ≠ n
  merge(n, m)

// Step 3
while ∃m ∈ made_smaller : m ∈ N  ∧  (all(c), m) ∈ E  ∧
                               Attrs(m) = {}  ∧  Objs(m) = {}
  shortcut(m)

// Step 4
∀p ∈ {p|(p, Class(all(c))) ∈ E}:
  remove_edge(p, Class(all(c)))
  ∀a ∈ {a|(c, a) ∈ aboves}:
     add_edge(p, a)
∀a ∈ {a|(c, a) ∈ aboves}:
  add_edge(a, all(c))

cleaned:=cleaned ∪ {n|n ≤ Class(all(c))}
done:=done ∪ {c}
```

The algorithm processes all library classes “bottom up”. Classes without subclasses are processed first, then classes whose subclasses have already been processed will be processed and `java.lang.Object` will always be the last class processed, as it is a superclass even to interfaces.

For each class c 4 steps are performed.

In Step 1 all classes below $Class(all(c))$ are visited. For each class all attributes and objects are traversed. An attribute is considered to belong to a class, if it represents a member of that class or a $check(T, M, pc, true)$ type-check against that class. Attributes that belong to class c are moved to $Class(all(c))$. If an attribute belongs to a superclass of c, a new node is created for that superclass and the attribute is moved there. These nodes are called *above* and will be integrated into the inheritance hierarchy later. Attributes belonging to subclasses of c or not belonging to any class (like $check(T, M, pc, false)$ attributes) remain at their current nodes. For objects the type of the object is checked. If the type is equal to c, the node is tagged for merging. If the type of is object is a superclass of c, the object must be an artificial object created to prevent two method definitions or class attributes from appearing at the same node, as no usual object with of type c' with $c \leq c'$ could have a table entry in the $all(c)$ column as this is not type-correct (e.g. an object of type `java.lang.Object` cannot access a member of `java.lang.String`). Finally an object can be a subclass of c. If the type is a non-library class, it is ignored, otherwise it must be a pointer

(as an object cannot be moved above its original type due to the access of constructors). These pointers are moved the same nodes used for attributes. This is only required for library classes that are `final`, but it seems strange if a pointer with a library type before the analysis gets a user-defined type afterward.

In Step 2 the classes marked in the previous step are all collapsed into $Class(all(c))$. This places e.g. all objects of type `java.lang.String` at the node containing all members of this class. In contrast to attributes, objects can only be moved up in the hierarchy by merging classes, as a simple move will not guarantee that the new class still contains all attributes accessed by the object. Here, this only applies to $check(T, M, pc, false)$ attributes (as all other attributes have already been moved to different classes), but of course these are important too.

Step 3 tries to remove any empty subclasses of $Class(all(c))$ that may have been created by moving attributes and objects.

Finally, step 4 integrates the nodes created in step 1 into the graph. All incoming edges to $Class(all(c))$ are "routed" through the new nodes. These nodes are temporary and will be removed during later iterations.

As this algorithm is hard to understand, an example will be given. Figure 8.1 shows a sample input graph for the algorithm. Each node is split into three parts: the name of node, its attributes and its objects. Attributes of a class are displayed simplified by an upper letter and a number, objects as lower letters and a number. For the original hierarchy inheritance relations were $c \leq b \leq a$, so the algorithm processes the classes in order c, b and a. Figure 8.2 shows the results of step 1 while processing class c. Members from nodes 5 and 6 are moved to 4 and the newly created special nodes. After the next step (figure 8.3), these nodes are merged into 4, moving $c1$ up in the process. After step 4, the special nodes are integrated into graph (figure 8.4). Now b is processed. Again, members are moved to new special nodes and 3. Before step 3 (figure 8.5), the special nodes from the previous steps are empty, but are marked for removal. The graph after b has been fully processed can be seen in figure 8.6, the final graph after a has been processed in figure 8.7.

The worst case complexity of the algorithm is $O(m * n)$, where n is the number of nodes in the graph and m the number of library classes. However, this is a very conservative approximation as the number of nodes n will decrease during execution of the algorithm and the calculation of *below* will traverse the whole graph only for `java.lang.Object`.

8.3 Automated Simplification

The concept lattices created by Snelting/Tip even for small programs are very fine grained and thus create huge class hierarchies. In many cases this

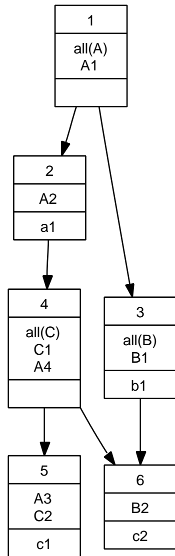


Figure 8.1: Initial hierarchy

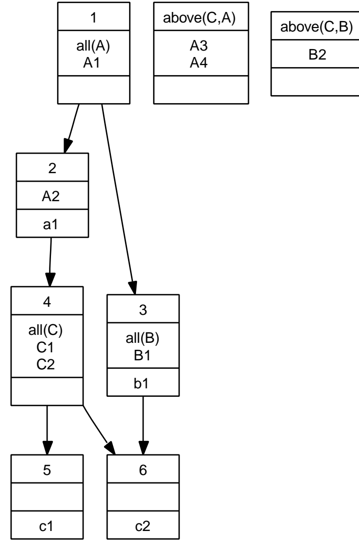


Figure 8.2: Processed C, step 1

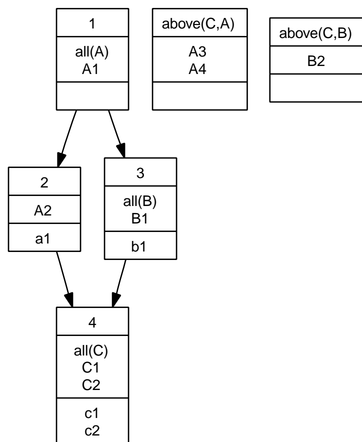


Figure 8.3: Processed C, step 2

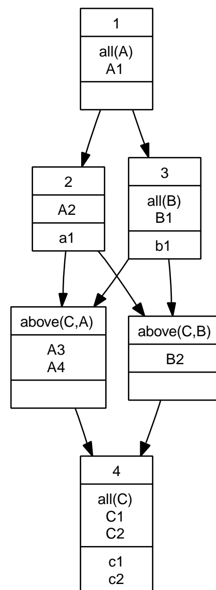


Figure 8.4: Processed C, step 4

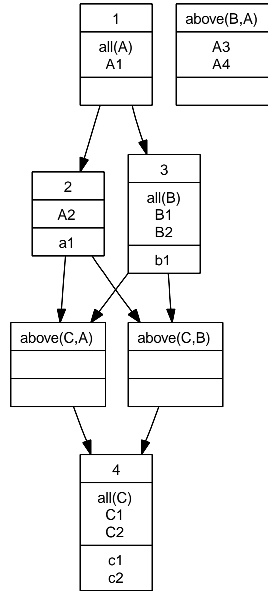


Figure 8.5: Processed B, step 2

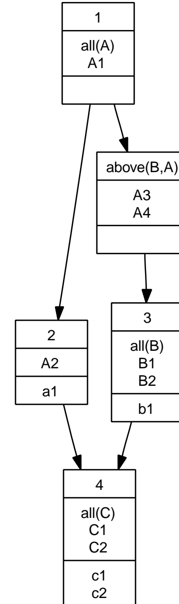


Figure 8.6: Processed B, step 4

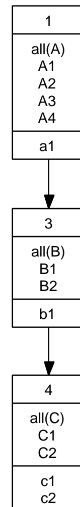


Figure 8.7: Processed A, step 4 (final)

level of granularity is not wanted. As making the hierarchy smaller by hand may be too time consuming, an automated method is needed.

The following algorithms are heuristic approaches that are based on observation of case studies. They are mainly based on structural properties of the graph and not necessarily on semantical information. While they perform "well" (especially if different algorithms are used together), if their quality is judged by the number of reduced classes, their modifications to the class hierarchy may be questionable from a semantical point of view. The decision, whether a given algorithm is suitable for application to a given class hierarchy, must still be made by the user.

These algorithms assume that library classes have been processed with the algorithm from the previous section. The main algorithm utilizes two smaller algorithms and helper functions, which are presented first.

```
modifiable(n) :=
   return ∃m : (m, n) ∈ n  ∧  ∄c : all(c) ∈ Attrs(n)
```

The helper function `modifiable` determines if class can be modified by the user. Library classes (recognizable by an $all(c)$ attribute) and the top element may not be modified.

```
merge_down_noobj(n) :=
  if not modifiable(n)
    return false
  next:={m|(n, m) ∈ E}
  if |next| ≠ 1
    return false
  if Objs(n) ∩ ClassVars(P) ≠ {}
    return false
  m:=only element of next
  merge(n, m)
  return true
```

The function `merge_down_noobj` tries to merge a class with its single subclass, if no instances of the class are created. This function is useful to remove interfaces and abstract classes. As it ignores instanced classes, it will not add additional members to objects and thus preserve the minimality of the class hierarchy for objects.

But even the number of classes which have instances may be too high and must be addressed. A second function takes care of this:

```
get_type(n) :=
  t:=⊥
  ∀a ∈ Attr(n):
    if a ≡ dcl(X.m)  ∨  a ≡ def(X.m)
```

```
      if t = ⊥
         t:=X
      else if t ≠ X
         return ⊥
  if t ≠ ⊥
    return t
  ∀a ∈ Attr(n):
    if a ≡ check(T, M, pc, true)  ∧  a checks against t'
      if t = ⊥
         t:='
      else if t ≠'
         return ⊥
  return t
```

```
merge_similar(n) :=
  prev:={m|(m, n) ∈ E}
  if |prev| ≠ 1
     return false
  merge:=false
  if ∀a ∈ Attr(n) : a ≡ class(T, M, pc, true)
    merge:=true
  if get_type(n) ≠ ⊥  ∧  get_type(n) = get_type(m)
    merge:=true
  if not merge
    return false
  m:=the only element of prev
  if not modifiable(m)
     return false
  merge(m, n)
  return true
```

`merge_similar` examines a class with a single superclass. If both contain only attributes for members from the same original class, they are merged. This includes attributes for member declarations and definitions, but also attributes for successful type-checks. If they are, the classes are merged, regardless of any objects or pointers they may have. This actually increases the objects size, as additional members are added to objects, but helps if a class got split into many small classes with no or only one member and lots of inheritance relations.

The actual simplification algorithm makes use of these two small algorithms.

```
simplify() :=
  todo := N
```

```
result:=false
while todo ≠ {}
  n := select one member of todo
  todo:=todo \ {n}
  try
    changed:=merge_down_noobj(n)
  success
    if changed
      result:=true
      todo := todo ∪ {m|(m,n) ∈ E ∨ (n,m) ∈ E}
  try
    changed:=merge_similar(n)
  success
    if changed
      result:=true
      todo := todo ∪ {m|(m,n) ∈ E ∨ (n,m) ∈ E}
return result
```

`simplify` traverses all nodes of the class hierarchy and applies the algorithms `merge_down_noobj` and `merge_similar` to all nodes. Nodes connected to modified nodes are later checked again, until a fix-point is reached. Neighbored nodes must be re-checked, as the algorithm may change the precondition of having a single sub-/superclass for the two called algorithms.

In practice this algorithms reduces the number of classes for class hierarchies with or without pointers greatly. Detailed numbers can be found in the case studies section (10.5).

8.4 Removing Multiple Inheritance

Multiple inheritance is the general term used, if a class is allowed to inherit from two different classes. In contrast to C++, Java supports only a limited version of multiple inheritance. Java distinguishes between classes and interfaces, where interfaces may not contain fields[1] and implementations (member definition). Each class can have only one superclass, but is not limited in the number of interfaces it implements.

Due to the minimality property of the concept analysis, the initial class hierarchy can easily contain multiple inheritance even for member definitions. Figure 8.8 shows and example program and figure 8.9 the resulting class hierarchy. Without multiple inheritance the method definitions for `f` and `g` must either be in the same class or in two classes with a super-/subclass relation. In all cases at least one object would have an additional

[1]They many contain fields for constants

```
 1  class A {
 2     void f() { }
 3     void g() { }
 4  }
 5
 6  class Test {
 7     public static void main(String args[]) {
 8        A a1=new A();
 9        a1.f();
10
11        A a2=new A();
12        a2.g();
13
14        A a3=new A();
15        a3.f();
16        a3.g();
17     }
18  }
```

Figure 8.8: Example program for multiple inheritance

member (compared to the analysis result), which it does not require. So a minimal hierarchy cannot be built without multiple inheritance.

As the original Snelting/Tip analysis was written for C++, this was not a general problem, because the language allows multiple inheritance. In Java multiple inheritance for classes must be removed to get a class hierarchy, that can be realized. This can be done by hand, but again an automation is desirable.

The first approach was to identify multiple inheritance "diamonds" in the class hierarchy and collapse them. While the identification is easy, the collapsing often caused the program semantics to be broken. It turned out, that a strategy which modifies the class hierarchy as less as possible is the most successful.

The core of the final algorithm is:

`move_members_up`(n) `:=`

$\forall m : n \leq m :$

$p(m) := p(m) \cup \{a | a \in Attr(m) \ \wedge \ a \equiv def(X.m)\}$

$\forall (n_1, a_1) \in \ p :$

$\forall (n_2, a_2) \in p : n_1 \neq n_2 \ \wedge \ n_1 \not\leq n_2 \ \wedge \ n_2 \not\leq n_1$

$above := \{n | n_1 \neq n\} \cap \{n | n_2 \neq n\}$

$above := above \setminus \{a | a \in above \ \wedge \ \exists b \in above : b \leq a\}$

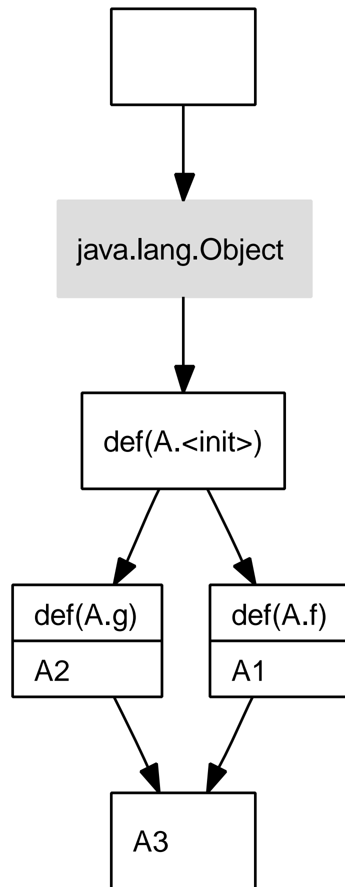


Figure 8.9: Class hierarchy for figure 8.8

$\forall n \in above$:`modifiable`(n)
  `try`
    $\forall a \in a_1$
      `move_attribute`(n_1, a, n)
  `success`
    `return` *true*
`return` *false*

This algorithm does not even change any inheritance relations. Instead it identifies pairs of nodes that are causing a class to need multiple inheritance, because both provide needed member definitions and are not in an inheritance relationship themselves. Then, affected members from one of these nodes are moved to one of the lowest common ancestor of these two nodes.

A simple example will illustrate how this works. Figure 8.10(a) shows a

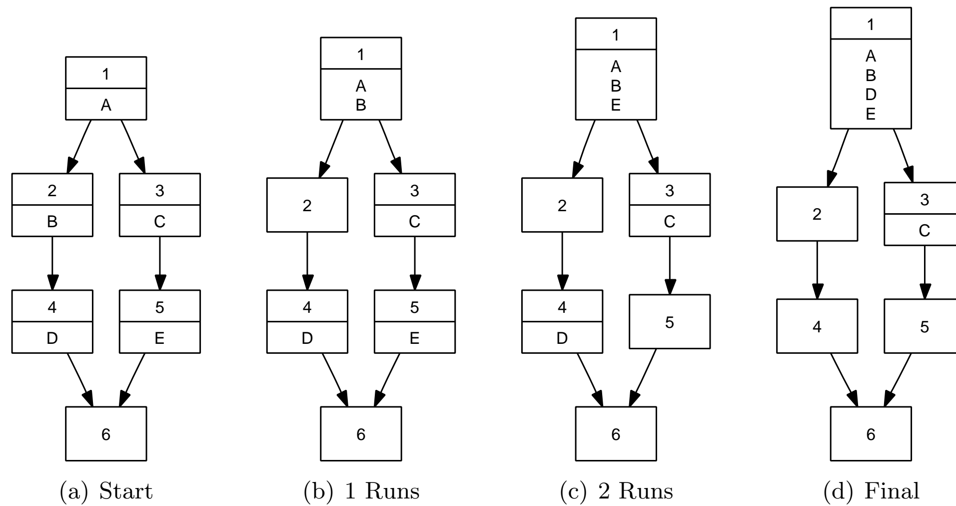


Figure 8.10: Multiple inheritance removal

class hierarchy, where the numbers work as node identifiers and the letters represent member definitions. Class 6 requires multiple inheritance and the function is started for this class. The following node pairs will reach the inner loop: $(2, 3)$, $(2, 5)$, $(4, 3)$, $(4, 5)$, $(3, 2)$, $(5, 2)$, $(3, 4)$ and $(5, 4)$. They are not checked in any particular order, so let us assume $(2, 3)$ is picked first. Then B will be moved to class 1, creating the hierarchy in figure 8.10(b). As class 6 still contains multiple inheritance, the function has to be run again, but this this time picking $(5, 4)$. The E is lifted to 1 (figure 8.10(c)). In the third run, only $(4, 3)$ or $(3, 4)$ remain as possible candidate, creating a hierarchy that might look like figure 8.10(d). The algorithm is not deterministic, but repeated application will always cause all members of classes 2 and 4 or 3 and 5 to be moved to class 1. If semantics prohibit moving of e.g. member E above member B, B will simply moved up first and E later, preserving this property at all times.

If the example, classes 2 and 4 are empty after application of the algorithm and can be merged into 1 or 6 to remove multiple inheritance completely. In real class hierarchies, they may not be empty, but can be still be removed by merging. Therefore the functions is always used together with simplification.

```
remove_multiple_inheritance() :=
  repeat:=true
  while repeat
    repeat:=false

    ∀n ∈ N
      while move_members_up(n)
```

```
        repeat:=true

    repeat:=repeat ∨ simplify()
```

Obviously it is possible to construct an example for multiple inheritance that cannot be resolved with this algorithm. But those examples do not seem to appear in practice as this algorithm has been able to remove the multiple inheritance in all cases ever tried. Probably an unresolvable form of multiple inheritance will never occur in a concept lattice because the original program can never contain it.

But still the algorithm leaves room for improvement. Currently it is very invasive, as it is coupled to the simplification algorithm. This is acceptable, as simplification will be run in most cases anyway, but this should not be enforced. Another point is the non-determinism of the algorithm. The given example has six different solutions (assuming none is blocked by the semantic restrictions), but the algorithm will just produce one. It is possible to calculate all possible solutions and the check for the best. This assumes there is one best solution, but this there is no reasonable way to select one of two symmetrical solutions as “better”. Additionally, calculating all solutions still is very expensive for big class hierarchies.

8.5 Other Algorithms

This section presents some more algorithms that were useful at one point or the other. They are not used in an automated process currently, but still available to a user for manual invocation.

All examples shown in this work have been processed with the algorithms `remove_empty` and `merge_down_this_only` in a fix point iteration.

8.5.1 Removing Empty Classes

```
remove_empty(n)=
  if Attrs(n) = {} ∧ Objs(n) = {}
    shortcut(n)
```

The most basic algorithm is `remove_empty`. It removes a class that has no objects or attribute by directly connecting the sub- and superclasses. Empty classes will most likely be contained in the initial class hierarchy created from the concept lattice and may be produced later as a by-product of moving attributes and objects around in the class hierarchy.

8.5.2 Removing Type-Check Nodes

For objects that will fail type-checks at run-time, $check(T, M, pc, false)$ attributes were introduced in order to make these objects distinguishable from

objects that will never be checked at run-time. Depending on programming-style and precision of the analysis, the number of objects that access such attributes varies highly. These attributes usually appear alone at a class and are then inherited for all objects requiring it. Figure 8.11 shows an example program and figure 8.12 the resulting class hierarchy illustrating this phenomenon.

As these attributes do not actually represent code or will appear in re-generated source code, they can be duplicated within the class hierarchy without further consequences. The following algorithm can be used:

```
distribute_typechecks(n)=
  if Attrs(n) ≠ {}  ∧  Objs(n) = {}
     ∧ ∀a ∈ Attrs(n) : a ≡ check(T, M, pc, false)
    distribute(n)
```

This algorithm removes a node that consist only of failed type-check attributes and puts each of them at all subclasses. The effects for the example given above can be seen in figure 8.13, which has a much easier structure. For the initial class hierarchy, this transformation is guaranteed to preserve semantics, as the corresponding attribute indicating a successful typecheck will never be below the attribute multiplied. However, it is explicitly allowed to move it there, so a semantic check must be performed in general.

8.5.3 Merging Nodes with only a This-Pointer

Sometimes it might be useful, not to remove all pointers, but only `this`-pointers. In contrast to the original Snelting/Tip-analysis, `this`-pointers are not forced to be below their corresponding *def*-attribute and will be above it, sometimes creating an individual class. A class whose only object is a `this`-pointer is non a good candidate, because the static type of the `this`-pointer will always be that of the implementing class of the method (i.e. the class with the *def*-attribute). Some of those classes can be removed with the following algorithm:

```
merge_down_this_only(n) :=
  if not modifiable(n)
    return false
  next:={m|(n, m) ∈ E}
  if |next| ≠ 1
    return false
  if ∃o ∈ Objs(n) : o is no this-pointer
    return false
  m:=the only element of next
  merge(n, m)
  return true
```

```
1   class A {
2   }
3
4   class B {
5   }
6
7   class C {
8   }
9
10  class D {
11  }
12
13  class Test {
14     static Object o;
15     static Object p;
16     public static void main(String args[]) {
17        o=new A();
18        o=new B();
19        o=new C();
20        o=new D();
21        p=new A();
22        p=new B();
23        p=new C();
24        p=new D();
25
26        if(o instanceof A)
27           o.hashCode();
28     }
29  }
```

Figure 8.11: Example program for type-check cause multiple inheritance

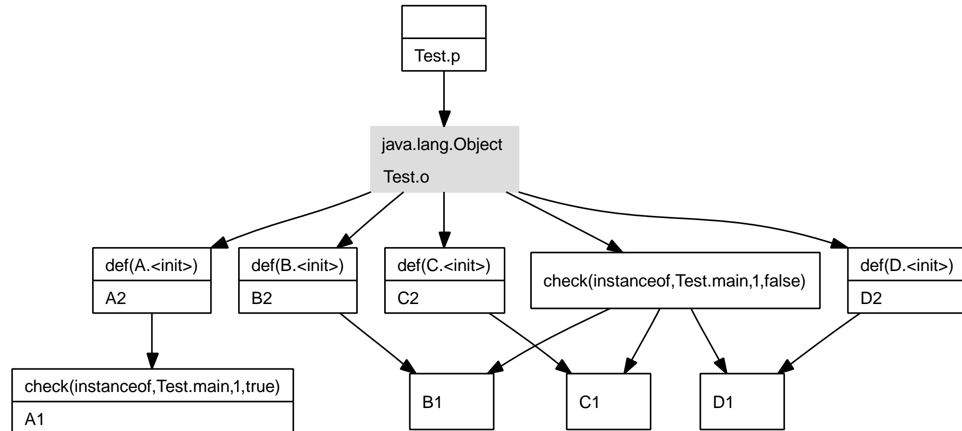


Figure 8.12: Class hierarchies for listing in figure 8.11

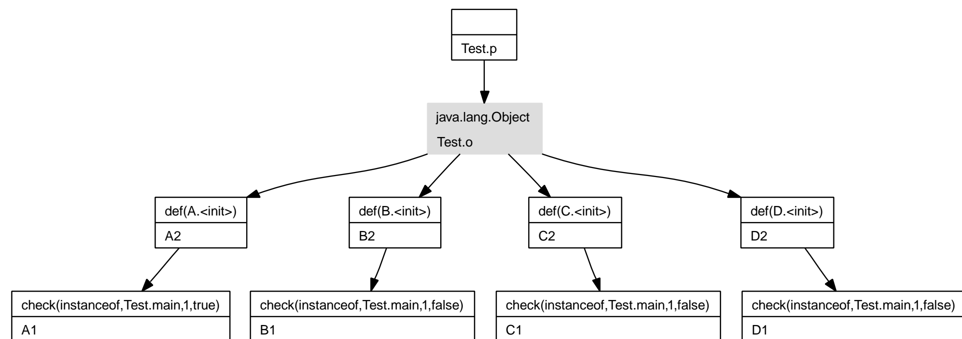


Figure 8.13: Result of `distribute_typecheck` on figure 8.12

This is a weaker version of `merge_down_noobj`. `merge_down_noobj` will merge more classes. It can be used if `merge_down_noobj` merges pointers, whose type actually reflects something in the source code (e.g. a parameter-pointer).

8.5.4 Merging Nodes Always

While `merge_down_this_ptr` is a weaker version of `merge_down_noobj`, stronger versions are also possible:

```
merge_down(n) :=
  if not modifiable(n)
    return false
  next:={m|(n,m) ∈ E}
  if |next| ≠ 1
    return false
  m:=the only element of next
  merge(n,m)
  return true

merge_up(n)=
  next:={m|(n,m) ∈ E}
  if |next| ≠ 1
     return false
  m:=the only element of prev
  if not modifiable(m)
     return false
  merge(m,n)
  return true
```

These algorithms will merge any class if it has a single sub- or superclass. This is a very invasive operation and should be used with care.

Chapter 9

KABA

9.1 Overview

KABA is the name of a prototype implementation of Snelting/Tip for Java. It consists of three parts:

- The static analysis for Java bytecode. It implements the analysis described in chapter 5.
- The dynamic analysis as an instrumentation of the JVM Kaffe. Its details are described in chapter 6.
- The graphical refactoring editor, which displays the refactoring proposal and allows interactive manipulation of the new hierarchy using the algorithms from chapter 8 while guaranteeing preservation of semantics using the techniques from chapter 7.

Two other projects contribute to KABA:

- KRS, the KABA Refactoring System. It was written by Peter Schneider[38] and provides a transformation of the original analyzed bytecode to the new class hierarchy.
- An alternate implementation of Snelting/Tip for Java was done by Thorsten Buckley[7] using SOOT[54] and SPARK[22]. It misses some features from KABA's own static analysis, but scales better and therefore can be used for larger programs.

9.2 The Graphical Refactoring Editor

The refactoring editor of KABA can be seen in figure 9.1. It shows the example from the introduction (figures 1.1 and 1.2). The class hierarchy from the introduction was slightly idealized to make it easier to understand,

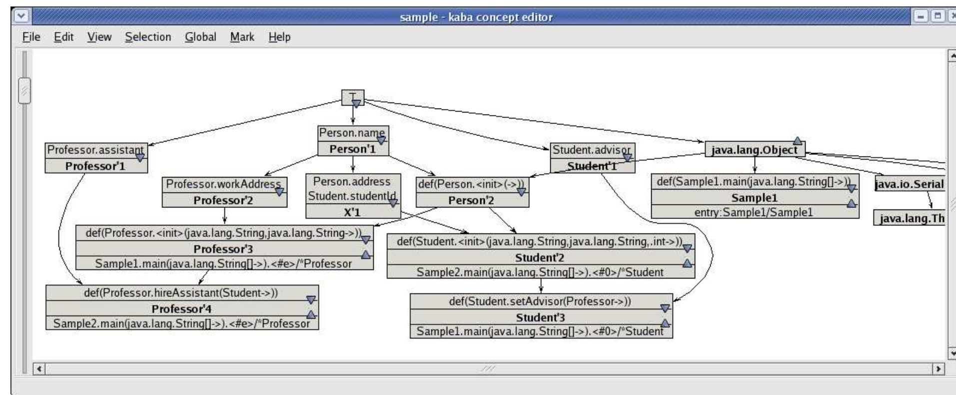


Figure 9.1: The KABA editor

whereas the editor shows the “real world” refactoring proposal. The most noticeable difference is the use of more complicated names for objects and full signatures in method names.

The notion is similar to the notion used in this thesis, but in the editor, every class already has an automatically proposed name (displayed in bold font). Members of the class are shown above this name, instances and pointers of that class below it. Class members are not shown by default. Each class node can be expanded, so it members and instances become visible (in the screen shots the `Professor` and `Student` classes are already expanded, whereas $\top$ and `java.lang.Object` are not). This was done to reduce the amount of space required for displaying the class hierarchy - a common problem also encountered in UML tools, where printed versions of class hierarchies can easily take up multiple square meters if printed. The graph layouter *dynadot* from the *graphviz*[1] package is used to create the initial layout and for incrementally creating new layouts, if the class hierarchy is changed.

The algorithms presented in chapter 8 are available either from a the context menu for a class or from the a global menu. The architecture of the editor is open, new plugins with new algorithms can be added anytime. An example of modifying the graph using context menus can be seen in figure 9.2. Here, the user decides that the class containing only the attribute `Professor.assistant` (called `Professor'1`) is not a good candidate for a class. From the context menu he gets the option, to merge it with its only successor. The class hierarchy after selection of that option can be seen in figure 9.3. The class has vanished and the field is now contained in the former subclass `Professor'4`.

Modifying the class hierarchy is not only possible through the application

[1] `www.graphviz.org`

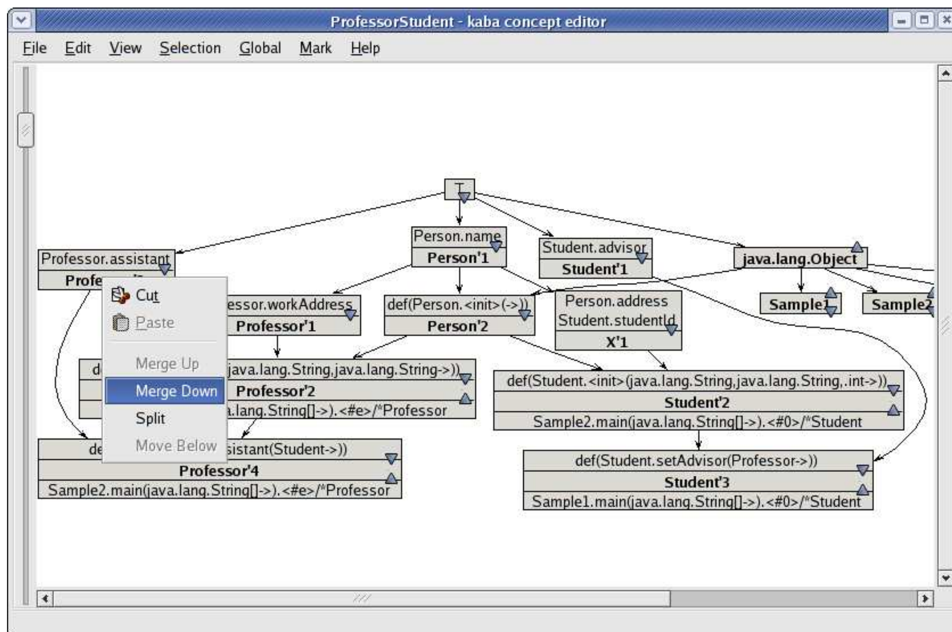


Figure 9.2: Modifying the class hierarchy in KABA - before

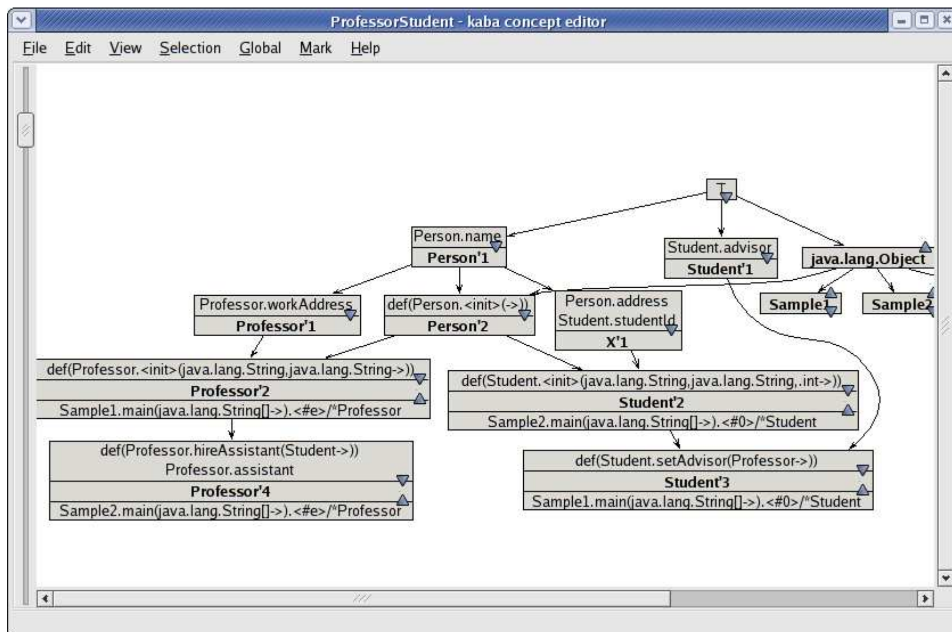


Figure 9.3: Modifying the class hierarchy in KABA - after

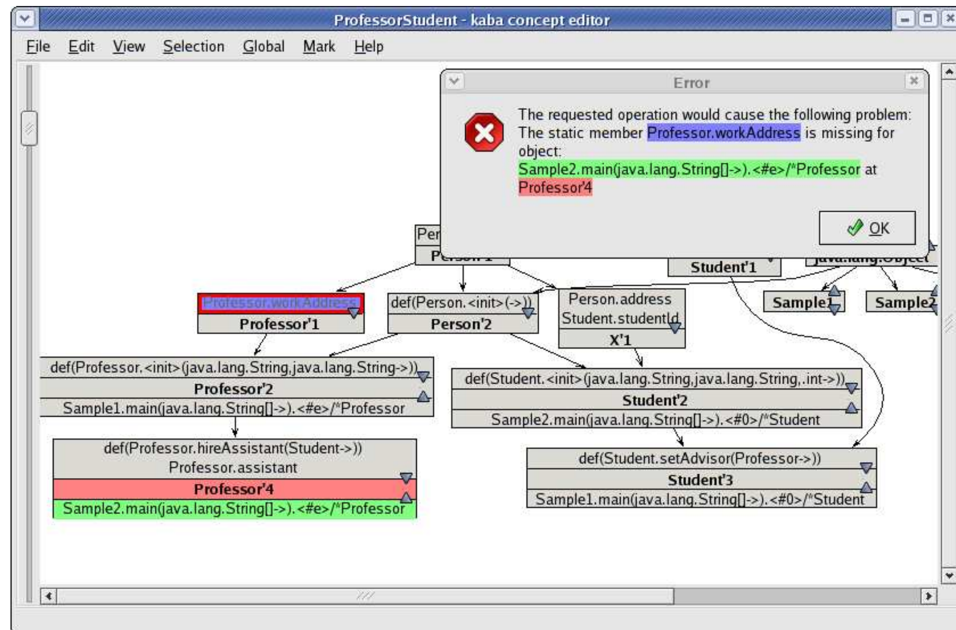


Figure 9.4: Error message from the editor

of algorithms. Basic operations like moving members or objects are available too. They can moved in a *cut-and-paste* fashion: They are cut from their original class and pasted to their new destination.

If a modification breaks the program semantics, it is refused by the editor. For every operation the user performs, the semantic checks presented in chapter 7 are done in the background. The user is provided with as much information as possible, on why his operation failed. An example can be seen in figure 9.4, where it was tried to move the member `Professor.workAddress` into one of the student classes. KABA will detect that this member is required by some of the objects. Within the error message names of classes, members and objects are highlighted in different colors to help user find them fast in the class hierarchy.

9.3 The KABA Refactoring System

Peter Schneider[38] implemented a tool that can do arbitrary class hierarchy transformations based on Java bytecode. The input is a class hierarchy in bytecode and a construction plan for a new hierarchy. It will produce a new hierarchy where members are moved according to the construction plan. Although usable without KABA, it is closely tied to it.

The generated bytecode is as close as possible to the bytecode the Java compiler creates, allowing to use bytecode decompilers to recreate even

source code. His tool implements his own variant of resolving multiple inheritance by linearizing inheritance relations.

KRS was tested with a test suite and real world programs like *JLex*. For this program it could be shown that the results of a program run for the transformed version are identical to those of the original version. But as the number of classes had increased, loading those classes took longer, which resulted in a performance degradation of factor 2. As *JLex* is a program with very short execution times, this is most likely an artifact of this example.

As a special bonus, KRS supports the same amount of reflection as KABA does and can even translate class names. Consider the following example:

```
class A {
   int f() { return 1; }
}
class B extends A {
   int f() { return 2; }
}
...
boolean b=...;
String s;
if(b)
   s="A";
else
   s="B";
Object o=Class.forName(s).newInstance();
((A)o).f();
```

As explained earlier, KABA is able to handle such simple forms of reflection. A valid refactored hierarchy for the classes `A` and `B` could look like:

```
interface AB {
   abstract int f() = 0;
}
class AB1 implemements AB {
   int f() { return 1; }
}
class AB2 implemements AB {
   int f() { return 2; }
}
```

To translate the usage of the two classes in the program fragment, the names have to be adjusted. KRS transforms the code into something similar to this:

```
...
boolean b=...;
String s;
if(b)
   s="A";
else
   s="B";
if(s.equals("A"))
   s="AB1";
else if(s.equals("B"))
   s="AB2";
Object o=Class.forName(s).newInstance();
((AB)o).f();
```

Although this is more an ugly hack that a real approach to the problems, it helps greatly for simple applications of reflections, e.g. the `.class` operator.

Chapter 10

Case Studies

The chapter presents case studies of real world programs that were analyzed with KABA. Instead of providing statistics for a huge number of programs, a few selected examples were chosen and are discussed in depth.

10.1 A "small" example

As an introductory example, *JLex* (version 1.2.5) was chosen. *JLex* is a scanner generator in the tradition of *lex* but for Java. The same example was used by other program analyzers before[34, 37]. It consists of 7823 lines of code and contains 26 classes, including one interface and three anonymous inner classes implementing this interface. Besides that it does not utilize inheritance. *JLex* was analyzed with the static analysis and the initial class hierarchy which is created from the concept lattice can be seen in figure 10.1. This hierarchy consists of 712 classes. A diamond shaped class indicates a class with members from a library class, while a box shaped class indicates a class with members from *JLex* itself. One class has an octagon shape, it contains members from *JLex* as well as the library. In this form, the hierarchies is not useful because classes from the library are not represented as single classes, but their members are spread to different classes. After the application of the algorithm `create_library_classes` from section 8.2 and

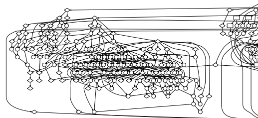


Figure 10.1: *JLex* 1.2.5, initial hierarchy

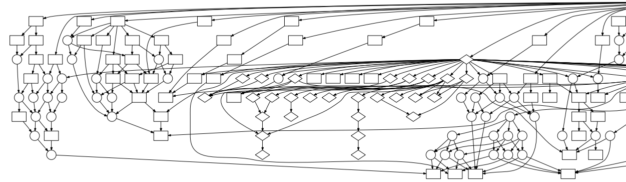


Figure 10.2: *JLex* after restoring system classes

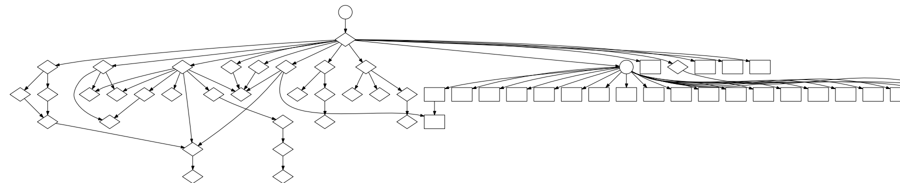


Figure 10.3: *JLex* after simplification

removing of unused system classes[1], the hierarchy gets much easier (seen in figure 10.2) and is reduced to 208 classes. The algorithm is automatically applied to all upcoming examples without further notice, the *raw* version was only presented here to give an idea about the size and complexity of the concept lattice. As creating 176 classes (32 classes are library classes) from original 26 still represents a level of detail that is questionable and probably will not be accepted as a refactoring proposal by most users. Therefore algorithms for further simplification have been developed. After applying the algorithm `simplify`(section 8.3), the class hierarchy is finally condensed (seen in figure 10.3) to a degree of complexity that is reasonable perceivable by a human (60 classes with 32 library classes). On the other hand, now the hierarchy is not much different from the original hierarchy. It seems that the simplification has not only reduced the complexity of the hierarchy, but also removed the gain of the Snelting/Tip algorithm. Closer inspection of *JLex* reveals, that except for three classes, all classes of *JLex* are only instantiated at one site in the program. Therefore for most classes only one access pattern is in the code and in the end, only one class will be produced. Having only a single instantiation site in the program is not uncommon, there are even popular design patterns (singleton, factory) which advocate this. This is of course a limitation of Snelting/Tip in practice, but fortunately most programs do not make such extensive use of this like *JLex* does. If single objects can create hierarchies as complex as those before simplification, the complexity is caused by access patterns from pointers. Although some useful information can be gained from these hierarchies (e.g. it can be used to get proposals for the *extract interface* refactoring), it was never a goal of Snelt-

[1]A system class is considered unused, if it is not sub-classed or instantiated outside of library code

Figure 10.4: *javac* after simplification

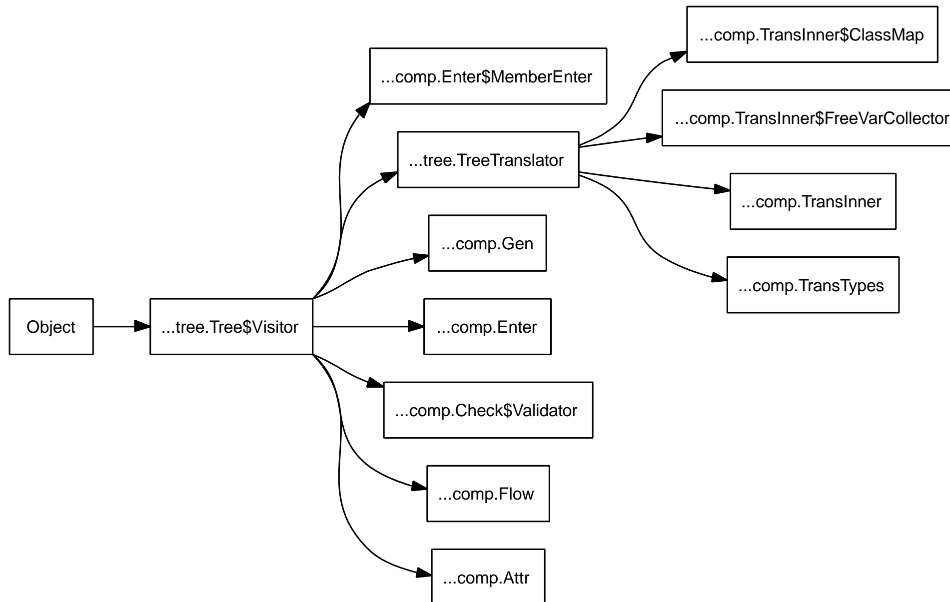


Figure 10.5: Class hierarchy for visitor pattern of *javac*

ing/Tip to create hierarchies of more than 25 abstract classes and interfaces for a single implementing class. Therefore, the following case studies ignore pointers to the greatest possible extent and focus on simplified hierarchies or use the dynamic analysis right from the start.

10.2 *javac*

The next example is Sun's Java compiler *javac* from the JDK 1.3. It consists of 145 classes and 28639 lines of code. The dynamic analysis was used and 1879 compilations of classes from JDK source code served as test-runs. The result after simplification can be seen in figure 10.4. As this hierarchy is still too big to display it completely, some interesting parts are broken out and discussed individually.

10.2.1 The Visitor Pattern

Figure 10.6 shows one sub-hierarchy of the original class hierarchy of *javac*, an implementation of the visitor pattern for the syntax tree. Package names have been omitted or are abbreviated. KABA's refactoring proposal can

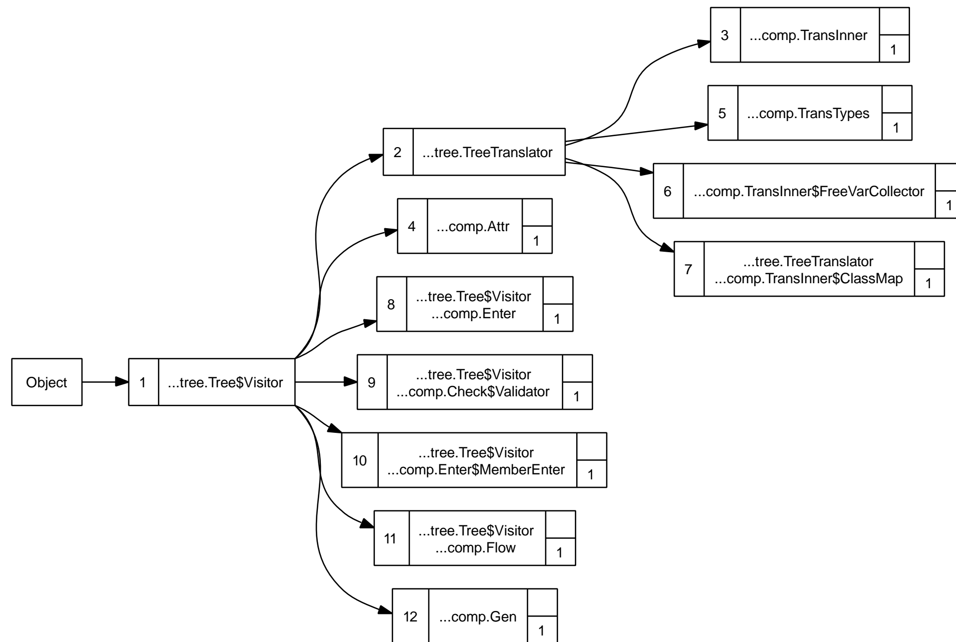


Figure 10.6: Refactoring proposal for figure 10.5

be seen in figure 10.6. Here all class have an additional number on the left side that is used for referencing them here in the text. The label of the class indicates from which original class a class contains members. E.g. classes 7 to 11 have members that were originally in two different classes and therefore the label has two lines. On the right side is the number of objects (lower box) or pointers (upper box, here always 0 as the dynamic analysis was used) that have this class as their type.

While the overall structure remains completely unchanged, some difference are visible. Some members from the superclass `Tree$Visistor`[2], have moved into different subclasses (8,9,10,11). This indicates they are only used from these subclass objects. Something similar can be seen for class 7, where a member of `TreeTranslator` was moved down. Moving these members down increases functional cohesion of these classes, as members that are used together are located in the same class. Of course, functional cohesion is not the only criteria for the quality of a class hierarchy, and in this case - the implementation of a well known design patterns - it may be more appropriate not to break the pattern to keep the code easily understandable.

A detailed inspection also showed, that a member of `Tree$Visistor` was detected as dead. As part of the verification of KABA, members presumed dead were removed from the source code, then the programs were compiled

[2]The $ in the name indicates that this class is in fact an inner class `Visitor` or the outer class `Tree`

and tested, if they were still working as expected. In this case, the compiler accepted the program without the method and then failed at run-time by throwing an exception and complaining about a missing implementation. What happened? The removal of the method changed the method selected at a call to an overloaded method. The exception and the error message it contains indicates that this behavior is intended and no subtle mistake. This looks like a dangerous practice that removing a dead method will break the program semantic and such practices should be avoided.

10.2.2 The Syntax Tree

The next example shows the hierarchy for the syntax tree (figure 10.7). The main class is `Tree` and it has subclasses for all syntactical units encountered in a Java program. This hierarchy is identically reproduced, indicating it was used in exactly the same way it was declared and cohesion for these classes is good and the interface of these classes was wisely chosen.

10.2.3 The Symbol Table

The last part of *javac* to be examined is the implementation of the symbol table. The original hierarchy can be seen in figure 10.8, it consists of 9 classes. The refactored version, shown in figure 10.9, has become slightly more complex, it consists of 17 classes. The top three subclass branches of the original superclass `Symbol` are almost identically reproduced, only the class `Symbol$MethodSymbol` has been horizontally split into three classes and a member of the original superclass was moved down. The fourth branch became slightly complicated. Here two subclasses were also split into multiple classes and additionally members of the original superclass are moved down too. Some of these members (in class 8) are accessed by objects of both subclasses, but not by all of these subclasses, creating a hierarchy where multiple inheritance is used to include these members only in classes that actually use them. In this case, multiple inheritance can be easily removed, by merging classes 8 and 13 with their superclass (6). There are other ways to achieve this goal, but it seems most natural, to resolve this by moving members up that were located in superclasses in the original hierarchy.

The proposed refactoring will again increase the functional cohesion and indicates a bit of redundancy in the original hierarchy, as it was possible to move down multiple members. Of course, this also improves information hiding, as fewer objects have access to these members. Whether this new hierarchy is more appropriate than the old, must be decided by the user of KABA. The proposed *split class* or *move method* refactorings are certainly justified if function cohesion is the only criteria, but a software engineer and user of KABA may decide otherwise. Therefore, KABA is an interactive system: The hierarchy it creates is just a proposal and there is room to

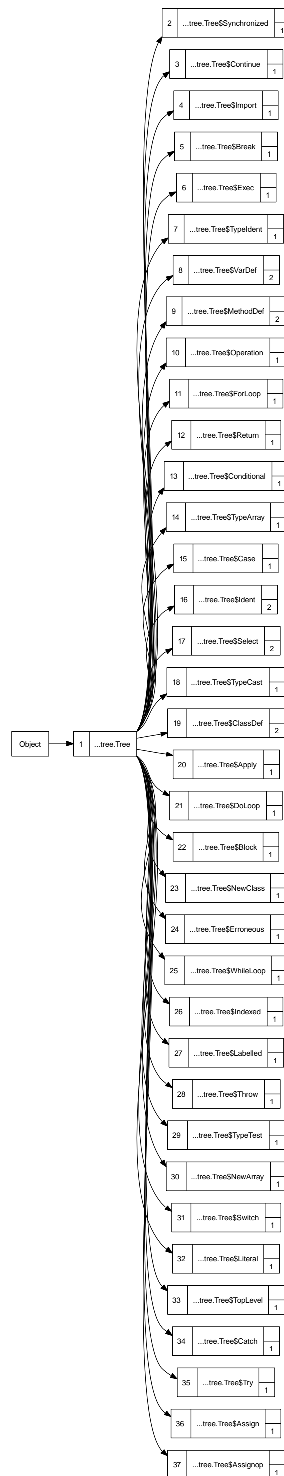


Figure 10.7: Class hierarchy for syntax tree in *javac*

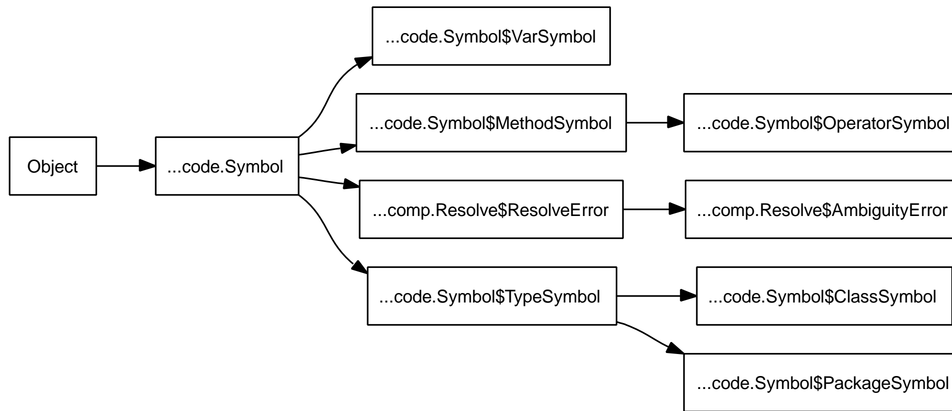


Figure 10.8: Class hierarchy for symbol table in *javac*

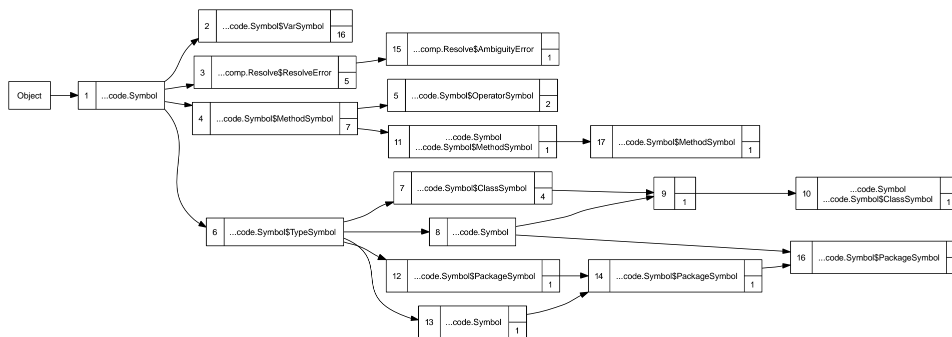


Figure 10.9: First refactoring proposal for figure 10.9

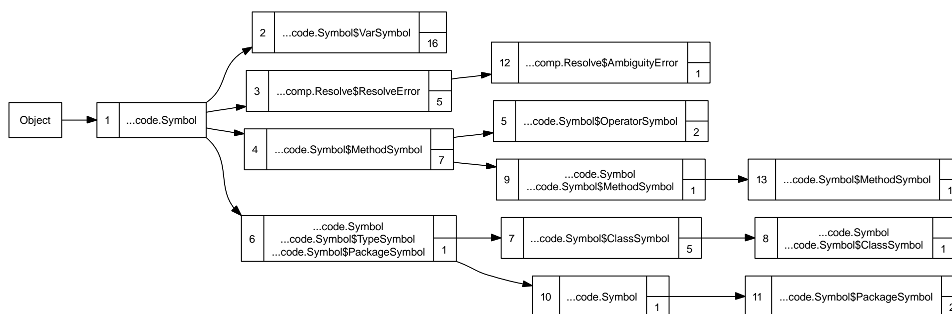


Figure 10.10: Second refactoring proposal for figure 10.9

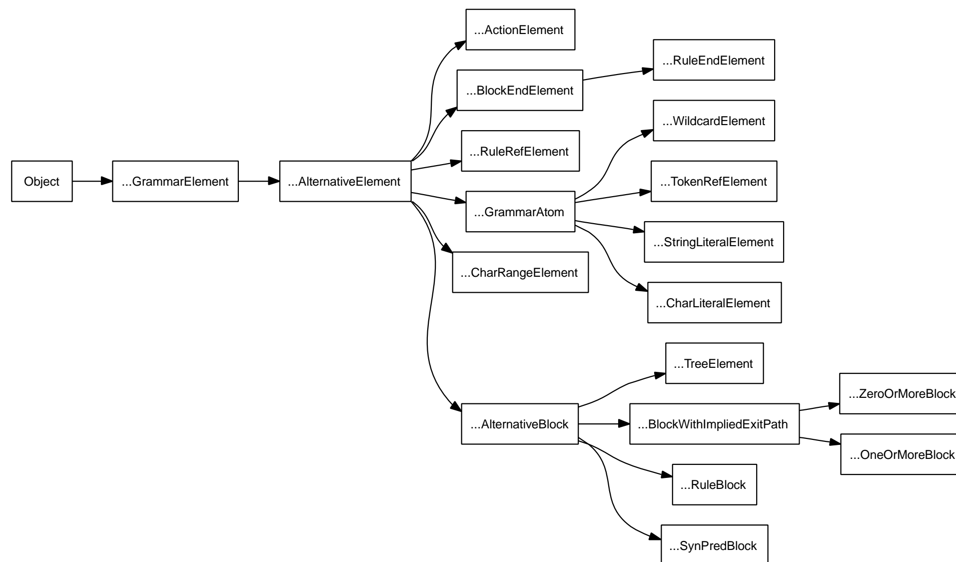


Figure 10.11: Class hierarchy for syntax tree of *antlr*

improve it. KABA provides the possibility to do so, without having to worry about the impact of the editing on the semantics of the program.

10.3 *antlr*

In all examples for the Java compiler the class hierarchy was modified only slightly. One could assume now, that this is always the case. The next case study shows that KABA can do more than that. The analyzed program is *antlr*, a scanner and parser generator for various languages. For the dynamic analysis all example grammars from *antlr* were used, creating 135 test runs. *antlr* has an additional run-time component for the created parsers that was not analyzed.

10.3.1 Syntax Tree

The original hierarchy for the implementation of the syntax tree can be seen in figure 10.11 and consists of 19 classes. The initial refactoring proposal can be seen in figure 10.12. Opposed to all previous examples, this hierarchy is really complicated and more than just a slightly different variation of the original version.

There are a number of things noticeable: Members of the original classes `GrammarElement` and `AlternativeElement` that were superclasses to all other classes before have been moved to 9 different classes! This indicates that functional cohesion for these classes was very bad. Members of these classes are still on top of the hierarchy, but merged into the same class,

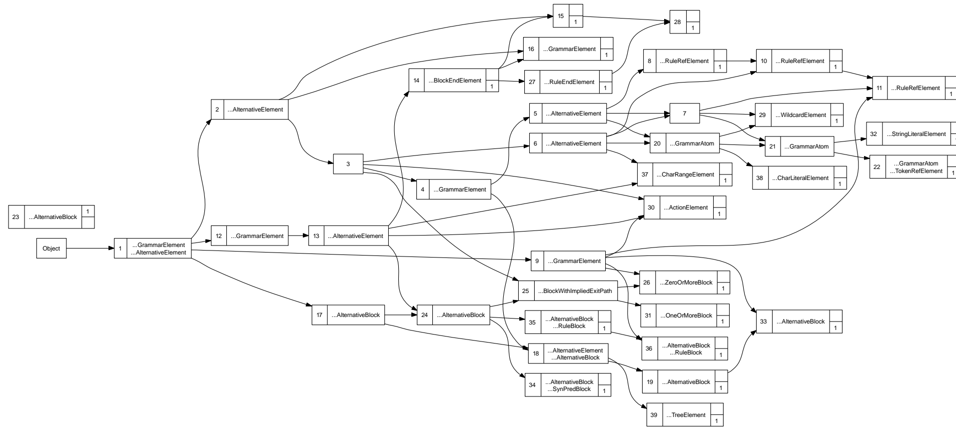


Figure 10.12: First refactoring proposal for figure 10.11

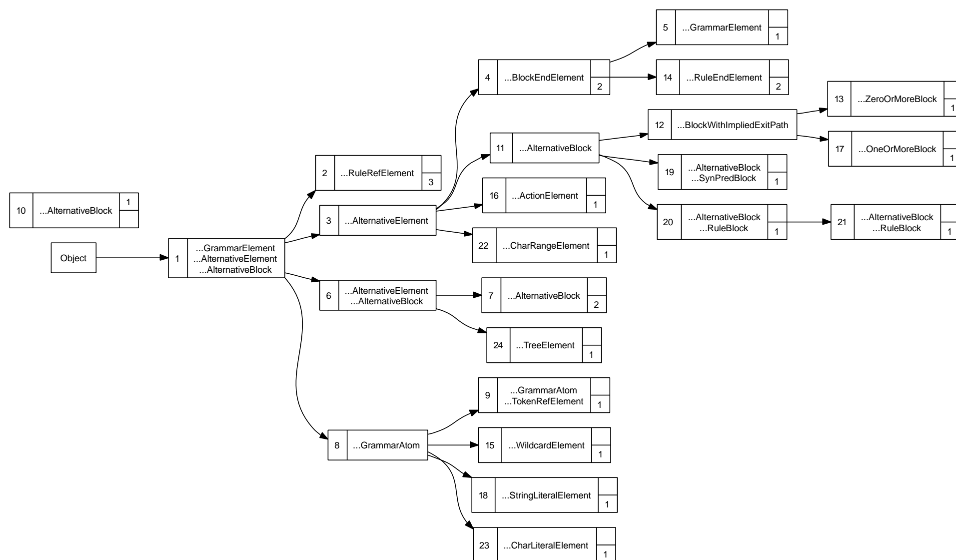


Figure 10.13: Second refactoring proposal for figure 10.11

Figure 10.14: Class hierarchy for tokens in *antlr*

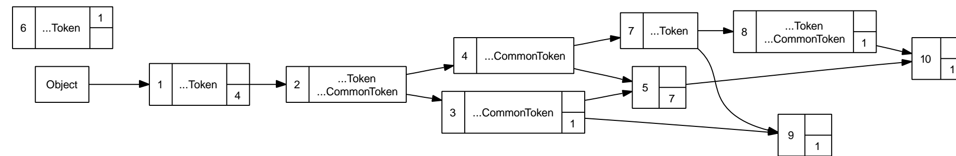


Figure 10.15: First refactoring proposal for figure 10.14 with dynamic analysis

which indicates the splitting of the two classes in the original hierarchy is questionable. The constructor of `GrammarElement` has been moved deep into the hierarchy at class 4.

Most other classes have been split as well. There is not a single class with more than 1 creation site showing that every object has an individual access pattern. `AlternateBlock`, originally a subclass of `AlternateElement` has partly been moved above that class, marking this inheritance relationship as questionable.

Class 29 is an isolated class, because it contains only a static member (the indicated pointer is used to create this artificial class). It can be merged with any other class in the hierarchy. Multiple inheritance is all over the class hierarchy and used in a more complicated way than see before: only 2 of 22 class that have instances can be realized without it. Removing multiple inheritance is required. Afterward (figure 10.13), the hierarchy looks much more smoothed and more similar to the original.

But still there are a lot of members moved and details changed. This restructuring brings a huge increase in functional cohesion.

Comparison with the syntax tree from *javac* shows a fundamental difference. Although both hierarchies represent similar functionality, there is a huge difference in the way their respective implementations are used. Without wanting to offend anyone, but *javac* seems a lot better designed. This indicates KABA can be used to evaluate a design: If there is no big difference between the way it is declared and the way it is used, it was a good design, otherwise something must be wrong with it.

10.3.2 Token

anltr is a program that besides the dynamic variant, can be analyzed by the static analysis of KABA. This gives the opportunity to compare the results of the static to those of the dynamic analysis. As an example the small

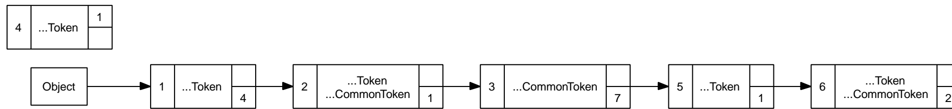


Figure 10.16: Second refactoring proposal for figure 10.14 with dynamic analysis

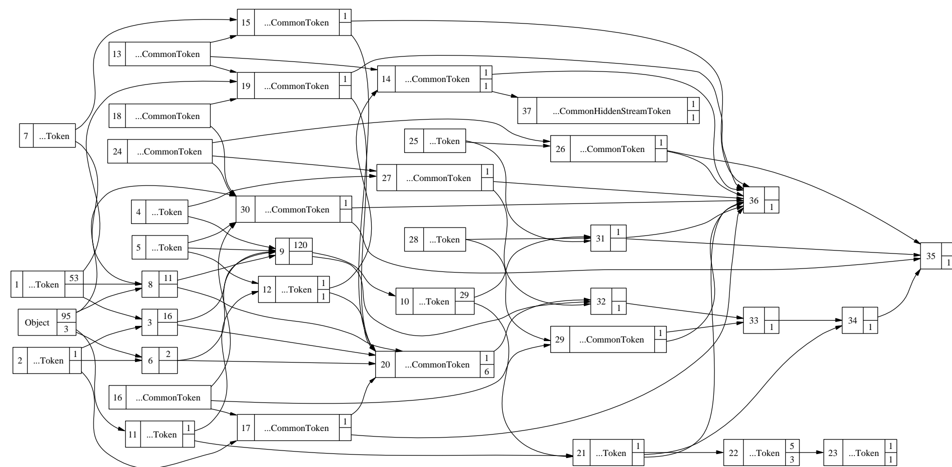


Figure 10.17: First refactoring proposal for figure 10.14 with static analysis

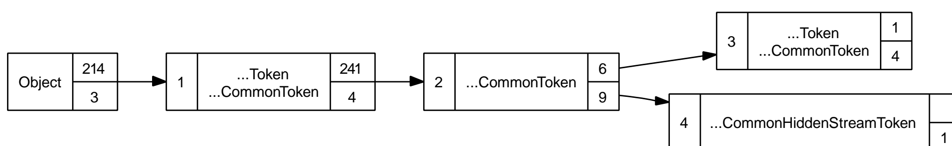


Figure 10.18: Second refactoring proposal for figure 10.14 with static analysis

objects	3	1	1	5	1	1	1	1	1	1
dynamic	1	1	3	5	5	5	6	8	9	10
static	22	23	20	20	32	33	11	36	34	35

Figure 10.19: Object types in static and dynamic analysis of *antlr*

hierarchy of classes representing parsed tokens was selected. The original hierarchy can be in figure 10.14.

The results of the dynamic analysis can be seen in figure 10.15. In the new hierarchy, members from the original classes `Token` and `CommonToken` are freely mixed in the hierarchy. Manual inspection reveals that the member moved to class 7 is `getType`, while `setType` is moved to class 8. However, the data field `type` they both access is in class 1 as it is initialized in the constructor of `Token`. Otherwise it could be moved down, too. Of course, if `setType` is moved to class 8, the instance of class 9 will always get the initial value for `type` as result of its calls to `getType`, show more potential for restructuring of this object. It is interesting to see that the whole token-type functionality is only accessed by objects from three instantiation sites, which is only a small part of the overall sites (14).

After the usual removal of multiple-inheritance (figure 10.16), the hierarchy becomes unbranched like the original class was, but is more fine grained. The original class `CommonHiddenTokenStream` does not appear in the new hierarchy and can therefore considered to be dead.

Of course, the result for the static analysis is a bit more complicated. It can be seen in figure 10.17. This is the same complexity already seen for the initial *JLex* example and is mainly caused by pointers. In this hierarchy, 10 classes are not below `java.lang.Object`. This indicates, that these pointers are not used to access members of `Object` and they cannot be instantiated either (an instance requires the constructor of `Object`). These classes make good candidates for interfaces if they do not contain data members. There are a few class with a high amount of pointers (1,9,10). They represent very common access patterns and thus serve as good starting points, if the class hierarchy is to be simplified by hand.

To compare the results of both analysis variants, the position of the different objects within the hierarchy must be compared. Figure 10.19 correlates these positions. Each column indicates a number of instantiation sites and their respective class number in figures 10.15 and 10.17. E.g. the first data column shows that objects from three different instantiation sites will have the type 1 in the dynamic and 22 in the static analysis. Surprisingly, there are objects (columns 3 and 4) where the dynamic analysis yields a higher level of detail, as these objects have the same type in the static, but different types in the dynamic analysis. There are also objects where the static analysis produces more detailed results (columns 4 to 6).

Assuming the number of test runs is high enough, the dynamic analysis is right and the higher details of the static analysis are caused by imprecision or conservativeness of the analysis.

The static analysis also contains the class `CommonHiddenStreakToken`, which was presumed dead after it did not appear in the results of the dynamic analysis. The obvious explanation would be a too small set of tests used during the dynamic analysis. Manual inspection reveals this is not the case! There is no creation site for this class in the source code of *antlr*, in fact class 37 is an artifact of KABA's approximation of reflection. *antlr* uses reflection (although not for the classes shown here), and at some point KABA assumes, an instance of that class is created.

If multiple inheritance is removed from the static version and some additional simplifications are performed in order to reduce the number of classes, the resulting class hierarchy is no longer more complex than the result of the dynamic analysis. It can be seen in figure 10.18. It is even coarse-grained than the final version of the dynamic analysis. Compared to the original hierarchy it still shows that the distinction between `Token` and `CommonToken` cannot be justified from the actual use of these classes and should be reconsidered. The hierarchies proposed by KABA give ideas how this can be done and KABA provides the infrastructure to try out different possibilities.

10.4 Varying the Number of Clients

Snelting/Tip makes a closed world assumption, i.e. it assumes all code using a class is known. In practice, this requirement may be too hard. It is possible that an unanalyzed client will work correctly with a restructured hierarchy, but of course there is no guarantee. But how likely is that? In order to get a feeling for this, a hierarchy was analyzed with a varying number of clients.

As a real hierarchy with a big number of clients for static analysis was not available, *KABA* itself was analyzed with the dynamic analysis was chosen and the number of test cases modified. 335 test cases were first analyzed individually. The sizes of input table and concept lattice are shown in figure 10.20. The test cases are spread among the x-axis the numbers of table rows, columns, initial entries and created classes can be seen on the y-axis.

With only a few exceptions, all test cases create tables and lattices of similar size. The noticeable range between 250 and 300 consists of a few structural tests that are very similar, so they all have an almost identical size for the input table. There are more groups of similar tests, but as they are not in any particular order, this is not well visible.

Figure 10.21 contains the *merged* tests: In analysis n (on the x-axis), the results from the first n test cases were joined into one table and analyzed. Of course, the number of table rows, columns and entries are constantly growing. While the number of classes in the hierarchy is generally rising

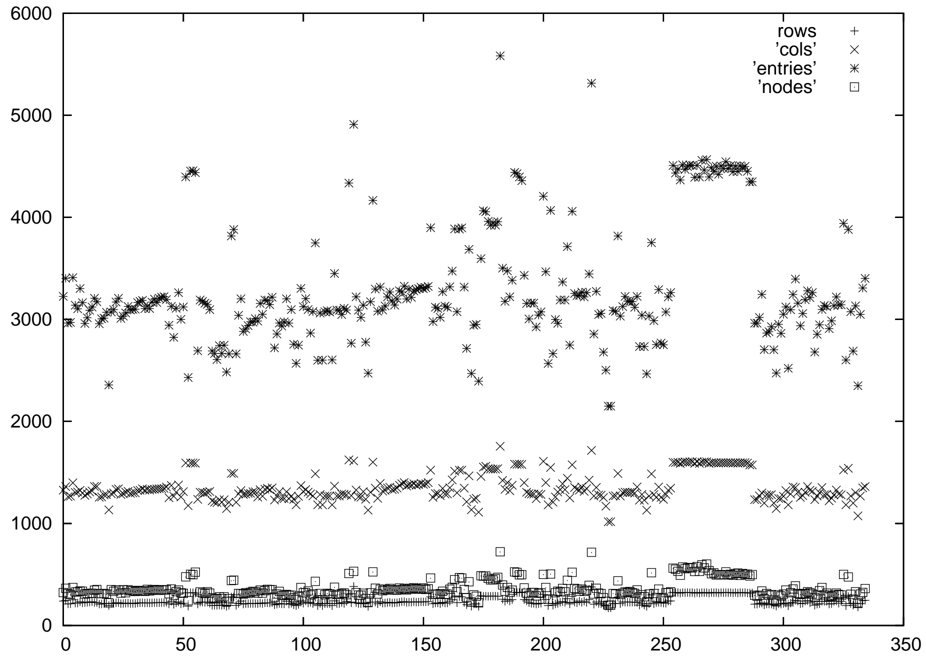


Figure 10.20: Sizes for individual test cases of *KABA*

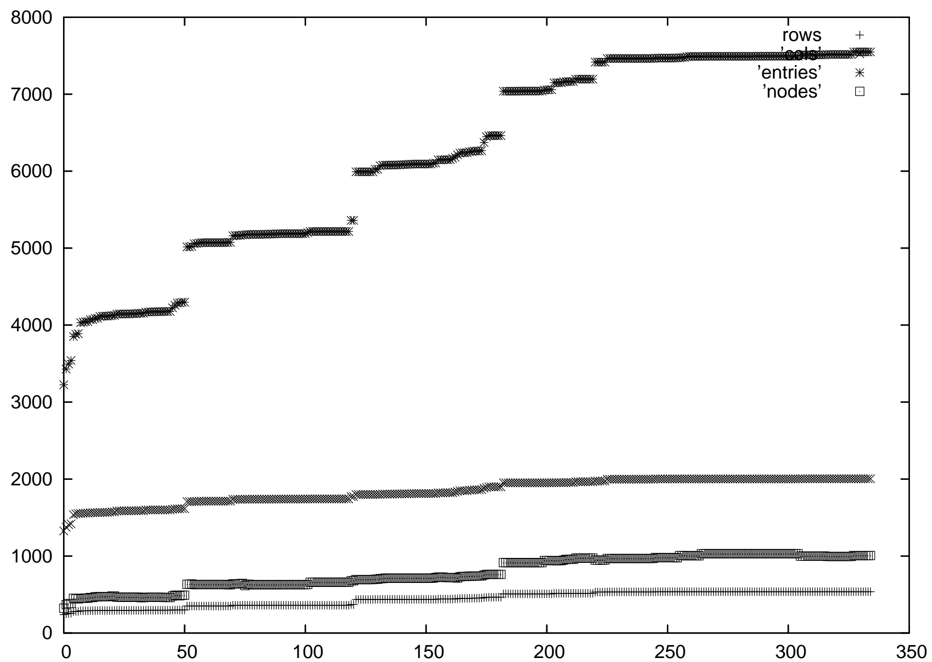


Figure 10.21: Sizes for merged test cases of *KABA*

there are several places, where it drops down a little bit (e.g. 20,75,220,310). This means an increased number of test cases can increase the complexity but can also decrease it (although an increase is the far more usual case).

A similar result is expected for a static analysis with a varying of clients, although there is a fundamental difference: For a dynamic analysis, the maximum number of table rows is static, for a static analysis it will vary, as every client brings in different table rows.

10.5 Performance

Figure 10.22 shows the running times of various steps of the analysis for the examples discussed in this chapter. The used machine is an Athlon 64 3200+, 2 GB RAM, Fedora Core 3 x86, JDK 1.5.0. The JVM was run using `-Xmx2000m`. The column *analysis* describes the analysis used: `S` for the static analysis, `D` for the dynamic analysis and `B` Buckley's alternate implementation using SOOT and SPARK. The column *JDK* gives the version of the JDK that was used. *Analysis time* sums up the whole cost of the static analysis. The number of table entries always refers to the initial table, before any propagation.

Initially the static analysis was a bottleneck, as it simply cannot analyze larger problems like *javac* in a reasonable amount of time. Buckley's implementation solves this problem and it does not seem likely, that the remaining features missing from his implementation will cause a serious slow-down.

But this implementation revealed the real bottleneck: the lattice construction, which was stopped after its running time exceeded 100 hours, even for an example that has a small table compared to others. As the algorithm used for lattice construction is the result of long research on the implementation of lattice construction algorithms, it seems questionable, whether there is room for improvement here. Please keep in mind that although this algorithm is $O(n^3)$ in practice, it has a theoretical worst case complexity of $O(2^n)$.

Analyzing the same example with multiple JDKs has revealed another problem: The size of the table highly depends on the JDK used, where the difference can be a factor of 6. As many Java programs already require newer JDKs than 1.3 (which is the base for the static analysis), this is a challenge for the future.

An alternative is the current dynamic analysis. It omit pointers and therefore creates much smaller tables and enables fast lattice construction.

The algorithms used on the class hierarchy after lattice creation are not particularly fast or slow. For bigger hierarchies they may be too slow for use in an interactive environment if the classical sub-seconds response time is required, but they are not a bottleneck. As these algorithms have undergone serious optimization work (speeding up the initial implementation by a

Example	# of classes	analysis	JDK	time					table size			# of classes		
				analysis	table iteration	lattice construction	library creation	simplification	rows	columns	entries	initial	after library creation	after simplification
JLex	26	S	1.3	18s	35s	14s	5s	4s	2957	1505	9355	712	208	60
JLex	26	B	1.3	59s	363s	>100h			17080	7871	438962			
javac	145	D	kaffe	-	0s	2s	1s	56s	306	2165	5679	509	499	147
javac	145	B	kaffe	141m	3603s	>100h			20037	8844	188723			
javac	145	B	1.3	165m	5037s	>100h			28252	12218	600443			
antlr	199	S	1.3	171m	2789s	10479s	27403s		16308	6726	162043	11619	7268	
antlr	199	D	kaffe	-	12s	1s	1s	22s	361	2379	5413	344	335	155
antlr	199	B	kaffe	67s	871s	>100h			12402	6122	115108			
antlr	199	B	1.3	94s	1376s	>100h			20626	9488	559637			
antlr	199	B	1.4	312s	2331s	>100h			32305	15360	1911744			

Figure 10.22: Performance data for all analyzes

factor of 1000) they probably do not leave that much room for performance improvement.

It should be noted that if context- and object-sensitivity could be enabled, running time for the static analysis would be greatly reduced. For the *antlr* example, enabling these techniques, along with an appropriate parametrization, reduces the analysis time by almost 90 percent!

Chapter 11

Discussion

The case studies have shown that KABA can uncover flaws in design and generate useful proposals on how to overcome this limitations. These proposals always increase functional cohesion. In some situations other aspects of design may be more important (e.g. some kind of redundancy may be required to build a consistent interface to a certain functionality) and this is the reason, why KABA's results are called proposals. In many cases they are not directly usable as a new class hierarchy, but give ideas where and how to modify the hierarchy in order to enhance it.

The initial proposals are often much too fine-grained, especially in the static analysis. This can be overcome with automatic simplification algorithms, but this is an heuristic based approach and there is no guarantee that these algorithms will not introduce the same flaws in the design, that had been uncovered by the analysis before. The first *antlr* example shows this in practice: KABA's initial proposal was totally different from the original hierarchy, but the automatically generated "final refactoring" is astonishingly similar to the original hierarchy, so it may in fact contain the same flaws again.

On the other hand, KABA provides the interactive refactoring editor, where a use can redesign the class hierarchy exactly as he wants and does not have to care whether his changes will break to program.

Unfortunately the case studies have also uncovered the limit of the analysis: Creation of the concept lattice may become very expensive for bigger software systems.

The new dynamic analysis does not suffer from this problem, as it generates much smaller tables due to the omission of pointers. The class hierarchies created that way are however no longer usable for an automatic program transformation, but only for proving ideas for possible refactorings. The detailed transformation (including pointers) must then be done by hand or by a different system.

Chapter 12

Future Work

To make KABA a tool suitable for a “normal” user, it must be integrated into a development environment (e.g. Eclipse). The approach of creating a whole new hierarchy will be to radical for some users, so it seems more appropriate to make KABA give refactoring proposals as modifications to the current hierarchy. The semantic model used for interactive refactoring could be a start for this: It depends only on information from the initial refactoring proposal and can then be used to check any possible class hierarchy for semantic correctness. A program could be analyzed, the semantic information built and then used to check if certain transformations in the unchanged hierarchy may be possible. Possibly the semantic information can even be built without creating a lattice at all, avoiding this bottleneck.

The scalability of the analysis is still an issue. The first approach would be not to analyze the whole program at once, but only smaller parts of the hierarchy (e.g. a single inheritance tree) as this will greatly reduce the resulting table. Some of the background for this is present in the way library classes are handled now, but additional work may be required, because the limitation that library class will not know the other classes must then be removed.

The most promising extension to KABA would be to extend the dynamic analysis so it provides enough information to allow automatic code transformation. Of course this will increase the size of the table and make creation of the concept lattice more expensive, but is not likely that it will reach tables as big as those of the static analysis, as usually the conservativeness of this analysis is the cause for a relevant part of the table. But in cases where the table generated by static and dynamic analysis will be almost identical, the lattice creation will be a limiting factor again.

For a dynamic analysis is seems also possible to handle the complete functionality of reflection and thereby remove the second limitation (besides the size) on programs that can be analyzed.

Appendix A

Constraints for the Bytecode Analysis

This appendix contains the contrains generated for all instructions of Java bytecode. For every instruction, all possible *RTE*, *LE* and *INIT* rules are listed first. These rules must be included unconditionally to the constraint system, if the corresponding instruction is included in the program. Then, the set constrains for stacks and local variables as well as other rules are listed. For a few instructions, there are multiple cases (e.g. `dup_x1`). As the preconditions for these cases are statically known, they are not added as conditional constrains. Instead only the rules for the given case are included into the constraint system.

- `aaload`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\bigwedge_{o \in S(M,C,pc,1)} CONTENT(o) \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `aastore`

$$RTE(M, C, pc, \texttt{java.lang.ArrayStoreException})$$
$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$

$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$

$$\bigwedge_{o \in S(M,C,pc,2)} S(M, C, pc, 0) \subseteq CONTENT(o)$$
$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `aconst_null`

$$\{NULL\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `aload` n

$$R(M, C, pc, n) \subseteq S(M, C, pc+2, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `aload_0`

$$R(M,C,pc,0) \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `aload_1`

$$R(M,C,pc,2) \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `aload_2`

$$R(M,C,pc,2) \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `aload_3`

$$R(M,C,p,3) \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `anewarray` t

$RTE(M, C, pc,$ `java.lang.NegativeArraySizeException`$)$

$LE(M, C, pc,$ `java.lang.IllegalAccessError`$)$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 3, 0)$$
$$type(OBJECT(M, pc, t)) = t\texttt{[]}$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$

- `areturn`

$RTE(M, C, pc,$ `java.lang.IllegalMonitorStateException`$)$

$$S(M, C, pc, 0) \subseteq RETURN(M, C)$$

- `arraylength`

$RTE(M, C, pc,$ `java.lang.NullPointerException`$)$

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `astore` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+2,i-1)$$
$$\bigwedge_{i=0}^{n-1} R(M,C,pc,i) \subseteq R(M,C,pc+2,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+2,n)$$
$$\bigwedge_{i=n+1}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+2,i)$$

- `astore_1`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$R(M,C,pc,0) \subseteq R(M,C,pc+1,0)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,1)$$
$$\bigwedge_{i=2}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `astore_2`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{1} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,2)$$
$$\bigwedge_{i=3}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `astore_3`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{2} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,3)$$
$$\bigwedge_{i=4}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `athrow`

$$RTE(M,C,pc,\texttt{java.lang.NullPointerException})$$
$$RTE(M,C,pc,\texttt{java.lang.IllegalMonitorStateException})$$

$$\bigwedge_{o \in S(M,C,pc,0)} THROW(M,C,pc,o,0)$$

- `baload`

$$RTE(M,C,pc,\texttt{java.lang.NullPointerException})$$
$$RTE(M,C,pc,\texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `bastore`

$$RTE(M,C,pc,\texttt{java.lang.NullPointerException})$$
$$RTE(M,C,pc,\texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\bigwedge_{i=3}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `bipush` n

$$\{INT\} \subseteq S(M,C,pc+2,0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+2,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+2,i)$$

- `caload`

$RTE(M, C, pc,$ `java.lang.NullPointerException`$)$
$RTE(M, C, pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `castore`

$RTE(M, C, pc,$ `java.lang.NullPointerException`$)$
$RTE(M, C, pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\bigwedge_{i=3}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `checkcast` t

$$RTE(M, C, pc, \texttt{java.lang.ClassCastException})$$
$$LE(M, C, pc, \texttt{java.lang.IllegalAccessError})$$

$$\bigwedge_{o \in S(M,C,pc,0)} type(o) \leq t \Rightarrow \{o\} \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `d2f`

$$\{FLOAT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `d2i`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `d2l`

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dadd`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `daload`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dastore`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dcmpg`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dcmpl`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `ddiv`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dload` n

$$\{DOUBLE\} \subseteq S(M, C, pc+2, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `dload_0`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dload_1`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dload_2`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dload_3`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dmul`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `dneg`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `drem`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dreturn`
  $RTE(M, C, pc,$ `java.lang.IllegalMonitorStateException`$)$

- `dstore` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+2, i-1)$$
$$\bigwedge_{i=0}^{n-1} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$
$$S(M, C, pc, 0) \subseteq R(M, C, pc+2, n)$$
$$\bigwedge_{i=n+2}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `dstore_0`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `dstore_1`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$R(M,C,pc,0) \subseteq R(M,C,pc+1,0)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,1)$$
$$\bigwedge_{i=3}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `dstore_2`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{1} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,2)$$
$$\bigwedge_{i=4}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `dstore_3`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{2} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$
$$S(M, C, pc, 0) \subseteq R(M, C, pc+1, 3)$$
$$\bigwedge_{i=5}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dsub`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dup`

$$S(M, C, pc, 0) \subseteq S(M, C, pc+1, 0)$$
$$S(M, C, pc, 0) \subseteq S(M, C, pc+1, 1)$$
$$\bigwedge_{i=1}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `dup_x1`

$$S(M,C,pc,0) \subseteq S(M,C,pc+1,0)$$
$$S(M,C,pc,1) \subseteq S(M,C,pc+1,1)$$
$$S(M,C,pc,0) \subseteq S(M,C,pc+1,2)$$
$$\bigwedge_{i=2}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `dup_x2`

$category(S(M,C,pc,1)) = 1 \Rightarrow$

$$S(M,C,pc,0) \subseteq S(M,C,pc+1,0)$$
$$S(M,C,pc,1) \subseteq S(M,C,pc+1,1)$$
$$S(M,C,pc,0) \subseteq S(M,C,pc+1,2)$$
$$\bigwedge_{i=2}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

$category(S(M,C,pc,1)) = 2 \Rightarrow$

$$S(M,C,pc,0) \subseteq S(M,C,pc+1,0)$$
$$S(M,C,pc,1) \subseteq S(M,C,pc+1,1)$$
$$S(M,C,pc,2) \subseteq S(M,C,pc+1,2)$$
$$S(M,C,pc,0) \subseteq S(M,C,pc+1,3)$$
$$\bigwedge_{i=3}^{S_{max}-1} S(M,C,pc,i) \subseteq S(M,C,pc+1,i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `dup2`

$category(S(M,C,pc,0)) = 1 \Rightarrow$

$$
\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0)\\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1)\\
S(M,C,pc,0) &\subseteq S(M,C,pc+1,2)\\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,3)\\
\bigwedge_{i=2}^{S_{max}-2} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+2)\\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}
$$

$category(S(M,C,pc,0)) = 2 \Rightarrow$

$$
\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0)\\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1)\\
\bigwedge_{i=1}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+1)\\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}
$$

- `dup2_x1`

$category(S(M,C,pc,0)) = 1 \Rightarrow$

$$
\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0)\\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1)\\
S(M,C,pc,2) &\subseteq S(M,C,pc+1,2)\\
S(M,C,pc,0) &\subseteq S(M,C,pc+1,3)\\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,4)\\
\bigwedge_{i=3}^{S_{max}-2} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+2)\\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}
$$

$category(S(M, C, pc, 0)) = 2 \Rightarrow$

$$
\begin{aligned}
S(M, C, pc, 0) &\subseteq S(M, C, pc + 1, 0) \\
S(M, C, pc, 1) &\subseteq S(M, C, pc + 1, 1) \\
S(M, C, pc, 0) &\subseteq S(M, C, pc + 1, 2) \\
\bigwedge_{i=2}^{S_{max}-1} S(M, C, pc, i) &\subseteq S(M, C, pc + 1, i + 1) \\
\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) &\subseteq R(M, C, pc + 1, i)
\end{aligned}
$$

- dup2_x2

$category(S(M, C, pc, 0)) = 1 \wedge category(S(M, C, pc, 2)) = 1 \Rightarrow$

$$
\begin{aligned}
S(M, C, pc, 0) &\subseteq S(M, C, pc + 1, 0) \\
S(M, C, pc, 1) &\subseteq S(M, C, pc + 1, 1) \\
S(M, C, pc, 2) &\subseteq S(M, C, pc + 1, 2) \\
S(M, C, pc, 3) &\subseteq S(M, C, pc + 1, 3) \\
S(M, C, pc, 0) &\subseteq S(M, C, pc + 1, 4) \\
S(M, C, pc, 1) &\subseteq S(M, C, pc + 1, 5) \\
\bigwedge_{i=4}^{S_{max}-2} S(M, C, pc, i) &\subseteq S(M, C, pc + 1, i + 2) \\
\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) &\subseteq R(M, C, pc + 1, i)
\end{aligned}
$$

$$category(S(M,C,pc,0)) = 2 \wedge category(S(M,C,pc,1)) = 1 \Rightarrow$$

$$\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0) \\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1) \\
S(M,C,pc,2) &\subseteq S(M,C,pc+1,2) \\
S(M,C,pc,0) &\subseteq S(M,C,pc+1,3) \\
\bigwedge_{i=3}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+1) \\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}$$

$$category(S(M,C,pc,0)) = 1 \wedge category(S(M,C,pc,2)) = 2 \Rightarrow$$

$$\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0) \\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1) \\
S(M,C,pc,2) &\subseteq S(M,C,pc+1,2) \\
S(M,C,pc,0) &\subseteq S(M,C,pc+1,3) \\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,4) \\
\bigwedge_{i=3}^{S_{max}-2} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+2) \\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}$$

$$category(S(M,C,pc,0)) = 2 \wedge category(S(M,C,pc,1)) = 2 \Rightarrow$$

$$\begin{aligned}
S(M,C,pc,0) &\subseteq S(M,C,pc+1,0) \\
S(M,C,pc,1) &\subseteq S(M,C,pc+1,1) \\
S(M,C,pc,0) &\subseteq S(M,C,pc+1,2) \\
\bigwedge_{i=2}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+1,i+1) \\
\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+1,i)
\end{aligned}$$

- `f2d`

$$\{DOUBLE\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `f2i`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `f2l`

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fadd`

$$\{FLOAT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `faload`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\{FLOAT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fastore`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fcmpg`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fcmpl`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fconst_0`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fconst_1`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fconst_2`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fadd`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fdiv`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fload` n

$$\{FLOAT\} \subseteq S(M, C, pc+2, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `fload_0`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fload_1`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fload_2`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fload_3`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fmul`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `fneg`

$$\{FLOAT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `frem`

$$\{FLOAT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `freturn`
  $RTE(M,C,pc,$`java.lang.IllegalMonitorStateException`$)$

- `fstore` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+2,i-1)$$
$$\bigwedge_{i=0}^{n-1} R(M,C,pc,i) \subseteq R(M,C,pc+2,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+2,n)$$
$$\bigwedge_{i=n+1}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+2,i)$$

- `fstore_0`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$S(M, C, pc, 0) \subseteq R(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fstore_1`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$R(M, C, pc, 0) \subseteq R(M, C, pc + 1, 0)$$
$$S(M, C, pc, 0) \subseteq R(M, C, pc + 1, 1)$$
$$\bigwedge_{i=2}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fstore_2`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{1} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$
$$S(M, C, pc, 0) \subseteq R(M, C, pc + 1, 2)$$
$$\bigwedge_{i=3}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `fstore_3`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{2} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$
$$S(M,C,pc,0) \subseteq R(M,C,pc+1,3)$$
$$\bigwedge_{i=4}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `fsub`

$$\{FLOAT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `getfield` $X.f$

$$RTE(M,C,pc,\texttt{java.lang.NullPointerException})$$
$$LE(M,C,pc,\texttt{java.lang.NoSuchFieldError})$$
$$LE(M,C,pc,\texttt{java.lang.IllegalAccessError})$$
$$LE(M,C,pc,\texttt{java.lang.IncompatibleClassChangeError})$$

$type(X.f)$ is a reference type $\Rightarrow$

$$\bigwedge_{o \in S(M,C,pc,0)} FIELD(o, LookupDefinition(X.f)) \subseteq S(M,C,pc+3,0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$\mathit{type}(X.f)$ is not reference type $\Rightarrow$

$$\{\mathit{type}(X.f)\} \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `getstatic` $X.f$

$$RTE(M, C, pc, \texttt{java.lang.NoClassDefFoundError})$$
$$LE(M, C, pc, \texttt{java.lang.NoSuchFieldError})$$
$$LE(M, C, pc, \texttt{java.lang.IllegalAccessError})$$
$$LE(M, C, pc, \texttt{java.lang.IncompatibleClassChangeError})$$
$$INIT(M, C, pc, X)$$

$\mathit{type}(X.f)$ is a reference type $\Rightarrow$

$$FIELD(\mathit{LookupDefinition}(X.f)) \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

$\mathit{type}(X.f)$ is not a reference type $\Rightarrow$

$$\{\mathit{type}(X.f)\} \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `goto` n

$$\bigwedge_{i=0}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `goto_w` n

$$\bigwedge_{i=0}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `i2b`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `i2c`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `i2d`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `i2f`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `i2l`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `i2s`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iadd`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `iaload`

$RTE(M,C,pc,$ `java.lang.NullPointerException`$)$
$RTE(M,C,pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `iand`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `iastore`

$RTE(M,C,pc,$ `java.lang.NullPointerException`$)$
$RTE(M,C,pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `iconst_m1`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `iconst_0`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `iconst_1`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `iconst_2`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iconst_3`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iconst_4`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iconst_5`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `idiv`

$$RTE(M, C, pc, \texttt{java.lang.ArithmeticException})$$

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `if_acmpeq` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `if_icmpne` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `if_icmplt` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `if_icmpge` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `if_icmpgt` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `if_icmple` n

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + n, i - 2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + n, i)$$

- `ifeq` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `ifne` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `iflt` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+n, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `ifge` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+n, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `ifgt` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `ifle` n

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+n,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+n,i)$$

- `ifnonnull` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+n, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `ifnull` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+n, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `iinc` i c

$$\bigwedge_{i=0}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+3, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `iload` n

$$\{INT\} \subseteq S(M, C, pc+2, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `iload_0`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iload_1`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iload_2`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `iload_3`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `imul`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `ineg`

$$\{INT\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `instanceof` t

$LE(M, C, pc,$ `java.lang.IllegalAccessError`$)$

$$\{INT\} \subseteq S(M, C, pc + 3, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$

- `invokeinterface` $X.m\ c\ n$

$RTE(M, C, pc,$ `java.lang.AbstractMethodError`$)$
$RTE(M, C, pc,$ `java.lang.UnsatisfiedLinkError`$)$
$RTE(M, C, pc,$ `java.lang.NullPointerException`$)$
$RTE(M, C, pc,$ `java.lang.IllegalAccessError`$)$
$RTE(M, C, pc,$ `java.lang.IncompatibleClassChangeError`$)$
$LE(M, C, pc,$ `java.lang.NoSuchMethodError`$)$
$LE(M, C, pc,$ `java.lang.IncompatibleClassChangeError`$)$

$type(TRETURN(X.m)) = VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 5, i - 1 - n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 5, i)$$

$$\bigwedge_{o \in S(M,C,pc,0)} CALL(M, pc, LookupDefinition(X.m, o), \{\}, \{o\}, S(M, C, pc, 1), \ldots, S(M, C, pc, n))$$

$type(TRETURN(X.m)) \neq VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+5,i-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+5,i)$$

$$\bigwedge_{o \in S(M,C,pc,0)} CALL(M,pc,LookupDefinition(X.m,o),S(M,C,pc+3,0), \{o\},S(M,C,pc,1),\ldots,S(M,C,pc,n))$$

- `invokespecial` $X.m$

$RTE(M,C,pc,$ `java.lang.UnsatisfiedLinkError`$)$
$RTE(M,C,pc,$ `java.lang.NullPointerException`$)$
$LE(M,C,pc,$ `java.lang.NoSuchMethodError`$)$
$LE(M,C,pc,$ `java.lang.AbstractMethodError`$)$
$LE(M,C,pc,$ `java.lang.IllegalAccessError`$)$
$LE(M,C,pc,$ `java.lang.IncompatibleClassChangeError`$)$

$type(TRETURN(X.m)) = VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$CALL(M,pc,X.m,\{\},S(M,C,pc,0), S(M,C,pc,1),\ldots,S(M,C,pc,n))$$

$type(TRETURN(X.m)) \neq VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$CALL(M,pc,X.m,S(M,C,pc+3,0),S(M,C,pc,0),\\ S(M,C,pc,1),\ldots,S(M,C,pc,n))$$

- `invokestatic` $X.m$

$RTE(M,C,pc,$`java.lang.NoClassDefFoundError`$)$
$RTE(M,C,pc,$`java.lang.UnstatisfiedLinkError`$)$
$LE(M,C,pc,$`java.lang.NoSuchMethodError`$)$
$LE(M,C,pc,$`java.lang.AbstractMethodError`$)$
$LE(M,C,pc,$`java.lang.IllegalAccessError`$)$
$LE(M,C,pc,$`java.lang.IncompatibleClassChangeError`$)$
$INIT(M,C,pc,X)$

$type(TRETURN(X.m)) = VOID \Rightarrow$

$$\bigwedge_{i=n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$CALL(M,pc,X.m,\{\},\{\},S(M,C,pc,0),\ldots,S(M,C,pc,n-1))$$

$type(TRETURN(X.m)) \neq VOID \Rightarrow$

$$\bigwedge_{i=n}^{S_{max}-max(1-n,0)} S(M,C,pc,i) \subseteq S(M,C,pc+3,i+1-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$CALL(M,pc,X.m,S(M,C,pc+3,0),\{\},$$
$$S(M,C,pc,0),\ldots,S(M,C,pc,n-1))$$

- `invokevirtual` m

$RTE(M,C,pc,$`java.lang.AbstractMethodError`$)$
$RTE(M,C,pc,$`java.lang.UnsatisfiedLinkError`$)$
$RTE(M,C,pc,$`java.lang.NullPointerException`$)$
$LE(M,C,pc,$`java.lang.NoSuchMethodError`$)$
$LE(M,C,pc,$`java.lang.AbstractMethodError`$)$
$LE(M,C,pc,$`java.lang.IllegalAccessError`$)$
$LE(M,C,pc,$`java.lang.IncompatibleClassChangeError`$)$

$type(TRETURN(m)) = VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$\bigwedge_{o \in S(M,C,pc,0)} CALL(M,pc,LookupDefinition(m,o),\{\},\{o\},$$
$$S(M,C,pc,1),\ldots,S(M,C,pc,n))$$

$type(TRETURN(m)) \neq VOID \Rightarrow$

$$\bigwedge_{i=1+n}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$$\bigwedge_{o \in S(M,C,pc,0)} CALL(M,pc,LookupDefinition(m,o),S(M,C,pc+3,0),\{o\},$$
$$S(M,C,pc,1),\ldots,S(M,C,pc,n))$$

- `ior`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `irem`

$RTE(M,C,pc,$`java.lang.ArithmeticException`$)$

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `ireturn`

$RTE(M,C,pc,$`java.lang.IllegalMonitorStateException`$)$

- `ishl`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `ishr`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `istore` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+2, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `istore_0`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `istore_1`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `istore_2`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `istore_3`

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `isub`

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `iushr`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `ixor`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `jsr` n

$$\{pc+3\} \subseteq S(M, C, pc+n, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+n, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `jsr_w` n

$$\{pc+5\} \subseteq S(M, C, pc+n, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+n, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+n, i)$$

- `l2d`

$$\{DOUBLE\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `l2f`

$$\{FLOAT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `l2i`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `ladd`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `laload`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `land`

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lastore`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.ArrayIndexOutOfBoundsException})$$

$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lcmp`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lconst_0`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lconst_1`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `ldc` c

$type(c) \in \{INT, FLOAT\} \Rightarrow$

$$\begin{array}{c} \{type(c)\} \subseteq S(M, C, pc+2, 0) \\ \bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i) \end{array}$$

$type(c) \in \{STRING\} \Rightarrow$

$$\begin{array}{c} \{OBJECT(M, pc, \texttt{java.lang.String})\} \subseteq S(M, C, pc+2, 0) \\ type(OBJECT(M, pc, \texttt{java.lang.String})) = \texttt{java.lang.String} \\ \bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i) \end{array}$$

- `ldc_w` c

$type(c) \in \{INT, FLOAT\} \Rightarrow$

$$\begin{array}{c} \{type(c)\} \subseteq S(M, C, pc+3, 0) \\ \bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i) \end{array}$$

$type(c) \in \{STRING\} \Rightarrow$

$$\{OBJECT(M, pc, \texttt{java.lang.String})\} \subseteq S(M, C, pc+3, 0)$$
$$type(OBJECT(M, pc, \texttt{java.lang.String})) = \texttt{java.lang.String}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `ldc2_w` c

$$\{type(c)\} \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `ldiv`

$$RTE(M, C, pc, \texttt{java.lang.ArithmeticException})$$

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lload` n

$$\{LONG\} \subseteq S(M, C, pc+2, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+2, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `lload_0`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lload_1`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lload_2`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- lload_3

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+1, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- lmul

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- lneg

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lookupswitch` *default npairs* $m_1, o_1, \ldots, m_{npairs}, o_{npairs}$

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + default, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + default, i)$$
$$\bigwedge_{t=1}^{npairs} \bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + o_{npairs}, i - 1)$$
$$\bigwedge_{t=1}^{npairs} \bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + o_{npairs}, i)$$

- `lor`

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lrem`

  $RTE(M, C, pc,$ `java.lang.ArithmeticException`$)$

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lreturn`

  $RTE(M, C, pc,$ `java.lang.IllegalMonitorStateException`$)$

- `lshl`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lshr`

$$\{INT\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lstore` n

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+2, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+2, i)$$

- `lstore_0`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lstore_1`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lstore_2`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lstore_3`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lsub`

$$\{LONG\} \subseteq S(M, C, pc + 1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 1, i - 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 1, i)$$

- `lushr`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `lxor`

$$\{LONG\} \subseteq S(M, C, pc+1, 0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `monitorenter`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `monitorexit`

$$RTE(M, C, pc, \texttt{java.lang.NullPointerException})$$
$$RTE(M, C, pc, \texttt{java.lang.IllegalMonitorStateException})$$

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `multianewarray` t n

$$RTE(M, C, pc, \texttt{java.lang.NegativeArraySizeException})$$
$$LE(M, C, pc, \texttt{java.lang.IllegalAccessError})$$

$$OBJECT(M, pc, t) \subseteq S(M, C, pc + 4, 0)$$
$$type(OBJECT(M, pc, t)) = t\texttt{[]}^n$$
$$\bigwedge_{i=n}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc + 4, i + 1 - n)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 4, i)$$

- `new` t

$$RTE(M, C, pc, \texttt{java.lang.NoClassDefFoundError})$$
$$LE(M, C, pc, \texttt{java.lang.InstantiationError})$$
$$LE(M, C, pc, \texttt{java.lang.IllegalAccessError})$$
$$INIT(M, C, pc, t)$$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 3, 0)$$
$$type(OBJECT(M, pc, t)) = t$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 3, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 3, i)$$

- `newarray` t

$$RTE(M, C, pc, \texttt{java.lang.NegativeArraySizeException})$$

$t = 4 \Rightarrow$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 2, 0)$$
$$type(OBJECT(M, pc, t)) = \texttt{boolean[]}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 2, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 2, i)$$

$t = 5 \Rightarrow$

$$\begin{aligned} \{OBJECT(M,pc,t)\} &\subseteq S(M,C,pc+2,0) \\ type(OBJECT(M,pc,t)) &= \texttt{char[]} \\ \bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+2,i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+2,i) \end{aligned}$$

$t = 6 \Rightarrow$

$$\begin{aligned} \{OBJECT(M,pc,t)\} &\subseteq S(M,C,pc+2,0) \\ type(OBJECT(M,pc,t)) &= \texttt{float[]} \\ \bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+2,i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+2,i) \end{aligned}$$

$t = 7 \Rightarrow$

$$\begin{aligned} \{OBJECT(M,pc,t)\} &\subseteq S(M,C,pc+2,0) \\ type(OBJECT(M,pc,t)) &= \texttt{double[]} \\ \bigwedge_{i=0}^{S_{max}-1} S(M,C,pc,i) &\subseteq S(M,C,pc+2,i+1) \\ \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) &\subseteq R(M,C,pc+2,i) \end{aligned}$$

$t = 8 \Rightarrow$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 2, 0)$$
$$type(OBJECT(M, pc, t)) = \texttt{byte[]}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 2, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 2, i)$$

$t = 9 \Rightarrow$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 2, 0)$$
$$type(OBJECT(M, pc, t)) = \texttt{short[]}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 2, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 2, i)$$

$t = 10 \Rightarrow$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 2, 0)$$
$$type(OBJECT(M, pc, t)) = \texttt{int[]}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 2, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 2, i)$$

$t = 11 \Rightarrow$

$$\{OBJECT(M, pc, t)\} \subseteq S(M, C, pc + 2, 0)$$
$$type(OBJECT(M, pc, t)) = \texttt{long[]}$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc + 2, i + 1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc + 2, i)$$

- `nop`

$$\bigwedge_{i=0}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `pop`

$$\bigwedge_{i=1}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `pop2`

$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `putfield` $X.f$

$RTE(M, C, pc,$ `java.lang.NullPointerException`$)$

$LE(M, C, pc,$ `java.lang.NoSuchFieldError`$)$

$LE(M, C, pc,$ `java.lang.IllegalAccessError`$)$

$LE(M, C, pc,$ `java.lang.IncompatibleClassChangeError`$)$

$type(X.f)$ is a reference type $\Rightarrow$

$$\bigwedge_{o \in S(M,C,pc,1)} \Rightarrow S(M,C,pc,0) \subseteq FIELD(o, LookupDefinition(X.f))$$

$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-2)$$

$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$type(X.f)$ is not a reference type $\Rightarrow$

$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-2)$$

$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

- `putstatic` $X.f$

$RTE(M,C,pc,$ `java.lang.NoClassDefFoundError`$)$

$LE(M,C,pc,$ `java.lang.NoSuchFieldError`$)$

$LE(M,C,pc,$ `java.lang.IllegalAccessError`$)$

$LE(M,C,pc,$ `java.lang.IncompatibleClassChangeError`$)$

$INIT(M,C,pc,X)$

$type(X.f)$ is a reference type $\Rightarrow$

$$S(M,C,pc,0) \subseteq FIELD(LookupDefinition(X.f))$$

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-1)$$

$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

$type(X.f)$ is not a reference type $\Rightarrow$

$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+3,i-2)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+3,i)$$

- `ret` n

$$\bigwedge_{t \in R(M,C,pc,n)} \bigwedge_{i=0}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,t,i)$$
$$\bigwedge_{t \in R(M,C,pc,n)} \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,t,i)$$

- `return`
  $RTE(M,C,pc,$ `java.lang.IllegalMonitorStateException`$)$

- `saload`
  $RTE(M,C,pc,$ `java.lang.NullPointerException`$)$
  $RTE(M,C,pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\{INT\} \subseteq S(M,C,pc+1,0)$$
$$\bigwedge_{i=2}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+1,i-1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+1,i)$$

- `sastore`
  $RTE(M,C,pc,$ `java.lang.NullPointerException`$)$
  $RTE(M,C,pc,$ `java.lang.ArrayIndexOutOfBoundsException`$)$

$$\bigwedge_{i=3}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i-3)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `sipush` n

$$\{INT\} \subseteq S(M, C, pc+3, 0)$$
$$\bigwedge_{i=0}^{S_{max}-1} S(M, C, pc, i) \subseteq S(M, C, pc+3, i+1)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+3, i)$$

- `swap`

$$S(M, C, pc, 1) \subseteq S(M, C, pc+1, 0)$$
$$S(M, C, pc, 0) \subseteq S(M, C, pc+1, 1)$$
$$\bigwedge_{i=2}^{S_{max}} S(M, C, pc, i) \subseteq S(M, C, pc+1, i)$$
$$\bigwedge_{i=0}^{R_{max}} R(M, C, pc, i) \subseteq R(M, C, pc+1, i)$$

- `tableswitch` *default low high* $t_1, \ldots, t_{high-low+1}$

$$\bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+default,i-1)$$

$$\bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+default,i)$$

$$\bigwedge_{t=1}^{high-low+1} \bigwedge_{i=1}^{S_{max}} S(M,C,pc,i) \subseteq S(M,C,pc+t_i,i-1)$$

$$\bigwedge_{t=1}^{high-low+1} \bigwedge_{i=0}^{R_{max}} R(M,C,pc,i) \subseteq R(M,C,pc+t_i,i)$$

Appendix B

Limitations of the Prototype

B.1 Late Addition of Assignments

KABA is split into different parts, most noticably the code analysis was seperated from the interactive parts. The constraint system is only present during analysis time, its results are stored in a condensed form which does not include all constraints, but only the relevant parts required to build the concept lattice.

The condensed analysis results still contain the list of assignments, but adding additional assignments to the list will no longer result in the checking of all conditional constraints, which may result in missing entries to the table (the same effect as seen for context- and object-senstive points-to analysis).

Currently KABA has no user interface to add individual assignments, as this would not provide anything, which can't be done with the interactive editor instead. Unfortunatly, adding assignments is the only reasonable way to enforce equal signatures for overriden methods (see section 3.6).

Fixing the cause of this problem is hard. Even if the code analysis and the interactive editor were merged into one program, enabling manual modification of the constraint system would require a new calculation of the concept lattice. But this causes a number of problems (response time of the interactive system, automated and very invasive changes to class hierarchy as the result of "small" changes by the user), which are propably not solvable in a system that is supposed to be usable by a programmer.

Fixing the effects of the different handling of assignments is a bit easier, although this possibility might seem a little grotesque. The current handling of additional assignments results in abstract classes, which have instances. Two solutions are possible: First, the user is notified of the problem and forced to refactor the class hierarchy in a way, that those classes are no longer abstract. Secondly, a dummy function, which is empty or throws an exception, is automatically inserted to make the class non-abstract. In both cases, semantics of the program will not be affected, as the guarantee, that

```
1   class A {
2      void f() { }
3      void g() { }
4   }
5
6   class Test {
7      static A p;
8      static A q;
9      public static void main(String[] args) {
10        p=new A();
11        p.f();
12        q=new A();
13        q.g();
14     }
15  }
```

Figure B.1: Sample program for assignment problem

the “missing” members are not invoked, is still given by the static analysis.

A small example will illustate this. Figure B.1 shows a small program, figure B.2 the analysis result. Let's now assume, the assignment `p=q` shall be added to the program. The constraint system would then re-evaulate the CALL rule and create a hierachy as seen in figure B.3. The interactive editor only updates the table, which results in the hierarchy as seen in figure B.4. In this hierarchy the object named `A2` contains an abstract declaration of `f()`, but no implementation. It should however be obvious, that adding a dummy method called `f()` will not modify the semantics of the original program.

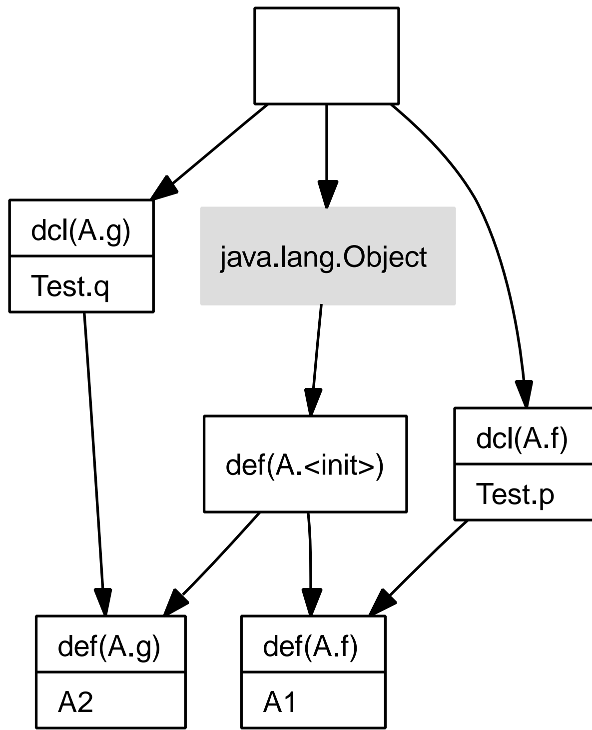


Figure B.2: Original analysis of B.1

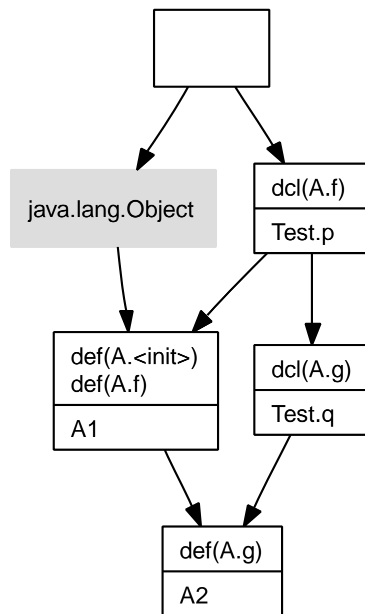


Figure B.3: Handling of `p=q` by the analysis

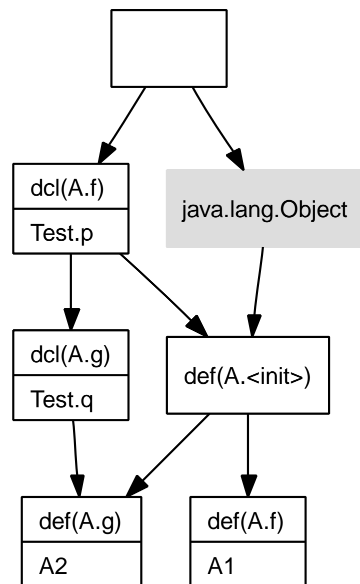


Figure B.4: Handling of `p=q` by the editor

Bibliography

[1] ISO/IEC 14882:2003.

[2] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2003.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[4] T. Bell. The concept of dynamic analysis. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234. Springer-Verlag, 1999.

[5] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114. ACM Press, 2003.

[6] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. Softw. Eng. Methodol.*, 7(2):109–157, 1998.

[7] T. Buckley. KABA als Fallstudie für das SOOT-Framework. Master's thesis, Universtität Passau, 2004.

[8] E. Casais. An incremental class reorganization approach. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 114–132. Springer-Verlag, 1992.

[9] U. Dekel. Applications of concept lattices to code inspection and review. In *The Israeli Workshop on Programming Languages and Development Environments*, chapter 6. IBM Haifa Research Lab, IBM HRL, Haifa University, Israel, July 2002.

[10] M. A. Fahndrich and A. Aiken. *Bane: a library for scalable constraint-based program analysis.* PhD thesis, University of California, Berkeley, 1999.

[11] S. J. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174. Springer-Verlag, 2000.

[12] J. S. Foster, M. Fahndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical report, University of California at Berkeley, 1997.

[13] M. Fowler. *Refactoring.* Addison-Wesley, 1999.

[14] M. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[16] B. Ganter and R. Wille. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag, 1999.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, second edition edition, June 2000.

[18] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105. ACM Press, 1998.

[19] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[20] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP*, pages 96–122, 2004.

[21] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, 2001.

[22] O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In G. Hedin, editor, *Compiler Construction: 12th International Conference*, 2003.

[23] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for java. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 73–79. ACM Press, 2001.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, second edition edition, 1999.

[25] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 349–359. ACM Press, 1997.

[26] C. Linding. *Algorithmen zur Begriffsanalyse und ihre Anwendung bei Softwarebibliotheken.* PhD thesis, Technische Universität Braunschweig, 8 1999.

[27] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11. ACM Press, 2002.

[28] I. Moore. Guru — A tool for automatic restructuring of self inheritance hierarchies. In R. Ege, editor, *Proceedings of TOOLS-USA'95, Santa Barbara, (CA), USA.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[29] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 235–250. ACM Press, 1996.

[30] I. Moore. *Automatic Restructuring of Object-Oriented Programs.* PhD thesis, Manchester University, 1996.

[31] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking:. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 11–20. ACM Press, 2002.

[32] W. F. Opdkye. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[33] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 66–73. ACM Press, 1993.

[34] J. Rajesh and D. Janakiram. Jiad: a tool to infer design patterns in refactoring. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 227–237. ACM Press, 2004.

[35] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java based on annotated constraints. Technical Report DCS-TR-428, Rutgers University, 11 2000.

[36] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated inclusion constraints. Technical Report DCS-TR-417, Rutgers University, 7 2000.

[37] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55. ACM Press, 2001.

[38] P. Schneider. Umsetzung von Transformationen an Klassenhierarchien in der Sprache JAVA. Master's thesis, Universität Passau, 2003.

[39] M. Siff and T. Reps. Identifying modules via concept analysis. In *Proc. of. the Internation Conference on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.

[40] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(2):146–189, 1996.

[41] G. Snelting. Concept lattices in software analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, 2003.

[42] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–110. ACM Press, 1998.

[43] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Trans. Program. Lang. Syst.*, 22(3):540–582, 2000.

[44] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[45] B. Steensgaard. Personal communication. Dagstuhl, November 1999.

[46] M. Streckenbach. Points-to-Analyse für Java. Technical Report MIP-0011, Fakultät für Mathematik und Informatik, Universität Passau, 2000.

[47] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330. ACM Press, 2004.

[48] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: reducing redundancies in inclusion constraint graphs. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–95. ACM Press, 2000.

[49] P. F. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 98–107. ACM Press, 2000.

[50] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 13–26. ACM Press, 2003.

[51] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293. ACM Press, 2000.

[52] F. Tip and P. F. Sweeney. Class hierarchy specialization. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–285. ACM Press, 1997.

[53] F. Tip and P. F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36:927–982, 2000.

[54] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[55] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255. IEEE Computer Society Press, 1999.

[56] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144. ACM Press, 2004.

[57] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999.