A Demand-Driven Solver for Constraint-Based Control Flow Analysis

Dissertation

Zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Diplom-Informatiker
Christian W. Probst

Saarbr¨ucken
September 2002

Tag des Kolloquiums:	4. Oktober 2002
Dekan:	Prof. Dr. Philipp Slusallek
Gutachter:	Prof. Dr. Reinhard Wilhelm Prof. Dr. Uwe Aßmann
Vorsitzender:	Prof. Dr. Andreas Zeller

Abstract

This thesis develops a demand driven solver for constraint based control flow analysis. Our approach is modular, flow-sensitive and scaling. It allows to efficiently construct the interprocedural control flow graph (ICFG) for object-oriented languages. The analysis is based on the formal semantics of a Java-like language. It is proven to be correct with respect to this semantics. The base algorithms are given and we evaluate the applicability of our approach to real world programs.

Construction of the ICFG is a key problem for the translation and optimization of object-oriented languages. The more accurate these graphs are, the more applicable, precise and faster are these analyses. While most present techniques are flow-insensitive, we present a flow-sensitive approach that is scalable.

The analysis result is twofold. On the one hand, it allows to identify and delete uncallable methods, thus minimizing the program's footprint. This is especially important in the setting of embedded systems, where usually memory resources are quite expensive. On the other hand, the interprocedural control flow graph generated is much more precise than those generated with present techniques. This allows for increased accuracy when performing data flow analyses. Also this aspect is important for embedded systems, as more precise analyses allow the compiler to apply better optimizations, resulting in smaller and/or faster programs.

Experimental results are given that demonstrate the applicability and scalability of the analysis.

Zusammenfassung

Diese Arbeit entwickelt einen Bedarf-gesteuerten L¨oser f¨ur Constraint-basierte Kontrollflußanalyse. Unser Ansatz ist modular, fluß-sensitiv and skaliert. Er erlaubt das effiziente Konstruieren des interprozeduralen Kontrollflußgraphen f¨ur objektorientierte Programmiersprachen. Die Analyse basiert auf der formalen Semantik einer Java-¨ahnlichen Sprache und wird als korrekt bez¨uglich dieser Semantik bewiesen. Wir pr¨asentieren die grundlegenden Algorithmen und belegen die Anwendbarkeit unseres Ansatzes auf realistische Programme.

Die Konstruktion des interprozeduralen Kontrollflußgraphen ist ein Schl¨usselproblem bei derÜbersetzung und Optimierung objekt-orientierter Programmiersprachen. Je genauer diese Graphen sind, desto pr¨aziser und schneller sind darauf arbeitende Datenfluß-Analysen. W¨ahrend die meisten heute verbreiteten Techniken fluß-insensitiv sind, pr¨asentieren wir einen skalierbaren fluß-sensitiven Ansatz.

Unsere Analyse hat zwei Hauptergebnisse. Einerseits erlaubt sie, nicht erreichbare Methoden zu identifizieren und zu l¨oschen, wodurch die Gr¨oße des erzeugten Programmes reduziert wird. Dies ist besonders f¨ur eingebettete Systeme wichtig, bei denen zus¨atzlicher Speicherplatz teuer ist. Andererseits ist der mit unserem Ansatz berechnete interprozedurale Kontrollflußgraph wesentlich genauer als der von derzeitigen Techniken berechnete Graph. Dieser pr¨azisere Graph erlaubt eine gr¨oßere Genauigkeit bei Datenflußanalysen. Auch dieser Aspekt ist f¨ur eingebettete Systeme von großer Bedeutung, da pr¨azisere Analysen bessere Optimierungen erlauben. Hierdurch wird das erzeugte Programm kleiner und/oder schneller.

Experimentelle Ergebnisse belegen die Anwendbarkeit und Skalierbarkeit unserer Analyse.

Extended Abstract

Programs written in powerful, higher-order languages [...] should run as fast as their `Fortran` *and* `C` *counterparts. They should, but they don't. A major reason is the level of optimisation applied to these two classes of languages.*

Olin Shivers, [Shi1991].

This thesis contributes a solution to one of the key problems in analyzing, optimizing, and translating object-oriented languages, namely the construction of the interprocedural control flow graph (ICFG). The solution is based on a demand driven solver for constraint systems generated by control flow analysis. Our approach is flow-sensitive and scalable. This is in contrast to the widely used techniques. Furthermore, by introducing functors into the constraint graph, the our approach allows to pre-analyze frequently used libraries.

The problem of ICFG construction occurs because object-oriented languages introduce the concept of *dynamic dispatch* where only at run-time of the compiled program it is decided which method shall be called. Therefore, at a call statement the compiler cannot easily determine the possible targets of the call. The well-accepted solution is to compute a conservative approximation of the ICFG. This results in two drawbacks.

On the one hand, an ICFG as precise as possible is of big importance for the accuracy of data flow analysis phases. In an ideal world, the ICFG would only contain those edges that really occur at run-time. Every additional edge between a call site and a method implies a potentially diluted result of the analysis, because imprecise call edges make the analysis investigate the code of the called method in a context that will not occur at run-time.

On the one hand, the construction of the ICFG identifies those methods that will never be called at run-time and thus may be deleted. Object-oriented languages and especially Java come with a huge collection of standard class libraries. The compilers we deal with always expect the *whole* program as input, i. e. the user program plus the libraries used. This approach is necessary, e. g., to allow application of interprocedural data flow analyzers. However, to compile an object-oriented program without any optimizations, one needs to compile all classes a program refers to. This may include lots of methods that will actually never be called. This is due to an immense interconnection between the classes in the standard class library. In order to translate, e. g.,

`HelloWorld.java`, what is an obligatory easy test case when writing compilers for imperative languages, one needs to be able to translate a big part of those libraries. The reason is, that the `main` method in a Java program gets an argument of type *String* and this class refers to lots of other classes. The number of classes to investigate for compilation of `HelloWorld` is 209 with 1629 methods.

As the standard libraries are so huge, constructing the ICFG usually allows to remarkably reduce the size of the created binary. As unreachable methods are deleted relatively early in the compilation chain, all following compiler phases that need to visit all methods in the program will be sped up. This includes, e. g., all data flow analyzers, optimizers, and code generation. The speed up is a strong reason to make the ICFG construction phase precise.

The analysis developed in this thesis is based on the well explored control flow analysis (CFA). This technique has been developed in the world of functional languages. Here, we adopt it to handle object-oriented languages. The analysis proposed is based on the operational semantics of `Bahasa`, a language that defines a subset of Java. We take a widely accepted Java semantics from literature as a basis for our analysis and prove the correctness of our results with respect to this.

An intermediate result when performing the analysis is a system of set-based inclusion constraints. They are converted to a directed graph, where the edges resemble inclusion. We develop a demand driven solver that benefits from special properties of this graph. It supports what we call *graph functors* that implement the call mechanism and reading and writing from class fields. Only at solving time we determine which methods to call and what fields to access, based on the intermediated results. When evaluating the functors, the solver may add new constraints to the graph.

Based on the computed results, call sites are annotated with the callable methods. From this information the ICFG is constructed.

The functors allow for another contribution. The standard class library is needed to compile any program, so one would like to pre-compile as much information as possible. The constraint graph is modular in the sense that it contains a sub-graph for every method in the program. Those sub-graphs are initially not connected, so we can for each method generate a description of the constraints and the graph. Due to the functors, these descriptions are completely independent of any other methods or run-time data and thus may be precomputed and saved. At compile-time they are combined with the graphs generated for the methods in the user program. This allows to speed up graph construction.

We give a small step structural operational semantics of the language used and prove the correctness of our analysis with respect to this. Additionally we will demonstrate its scalability by evaluating it on real world programs. This evaluation is performed with `JoC`, the compiler developed in the `JOSES` project. It has been funded by the European Community in the 4^{th} framework as a long time research project.

Ausführliche Zusammenfassung

> *Programme in Sprachen höherer Ordnung [...] sollten so schnell ausgeführt werden wie ihre in* `Fortran` *oder* `C` *geschriebenen Gegenstücke. Sie sollten, aber sie werden es nicht. Ein Hauptgrund hierfür sind die unterschiedlichen Optimierungen die auf diese beiden Sprachklassen angewandt werden.*
>
> *Olin Shivers, [Shi1991].*

Das Ergebnis dieser Arbeit ist eine L¨osung f¨ur eines der Kernprobleme bei der Analyse, Optimierung und Übersetzung objekt-orientierter Programmiersprachen – die effiziente Konstruktion eines m¨oglichst genauen interprozeduralen Kontrollflußgraphen (IKFG). Unser Ansatz basiert auf einem Bedarf-gesteuerten L¨oser f¨ur Constraint-basiert Kontrollflußanalyse. Der pr¨asentierte Ansatz ist fluß-sensitiv und skaliert. Dies ist der entscheidende Unterschied zu den heute weit verbreiteten Techniken. Durch den Einsatz von Funktoren erlaubt unser Ansatz die Vorabanalyse h¨aufig benutzter Bibliotheksfunktionen.

Die Konstruktion des IKFG ist problematisch, da objekt-orientierte Sprachen das Konzept des *dynamic dispatch* einf¨uhren. Erst zur Laufzeit des ¨ubersetzten Programmes wird entschieden, welche Methode ausgef¨uhrt wird. Daher kann der Übersetzer f¨ur einen Methodenaufruf nicht ohne weiteres die m¨oglichen Zielmethoden bestimmen. Die g¨angige L¨osung ist, eine konservative Absch¨atzung des IKFG zu berechnen. Hiermit sind jedoch Nachteile verbunden.

Einerseits ist ein m¨oglichst genauer IKFG von großer Bedeutung f¨ur Datenflußanalysen. In einer idealen Welt enthielte der IKFG nur Kanten, die Aufrufen zur Laufzeit entsprechen. Jede zus¨atzliche Kante zwischen einem Aufruf und einer Methode bedingt eventuell ein ungenaueres Ergebnis der Analyse, da aufgrund ¨uberfl¨ussiger Aufrufkanten die aufgerufene Methode in Kontexten analysiert wird, die zur Laufzeit nicht auftreten.

Andererseits werden bei der Konstruktion des IKFG unerreichbare Methoden identifiziert, die gel¨oscht werden k¨onnen. Objekt-orientierte Sprachen im allgemeinen und Java im besonderen verwenden große Standardklassenbibliotheken. Die von uns betrachteten Übersetzer erwarten immer das *ganze* Programm als Eingabe, das heißt das Benutzerprogramm und die verwendeten Bibliotheken. Dieser Ansatz ist notwendig um zum Beispiel interprozedurale Datenflußanalysatoren anwenden zu k¨onnen. Allerdings m¨ussen bei der Übersetzung eines unoptimierten

objekt-orientierten Programmes alle Klassen und unaufrufbare Methoden mit ¨ubersetzt werden. Dies liegt an der großen Verflechtung der Standardklassen untereinander. Um zum Beispiel `HelloWorld.java` zu ¨ubersetzen, muß man beinahe die vollst¨andigen Klassenbibliotheken ¨ubersetzen k¨onnen. Der Grund hierf¨ur ist, daß die `main` Methode eines Java Programms ein Argument vom Typ `String` erwartet. Diese Klasse verweist auf eine große Zahl weiterer Klassen, so daß f¨ur dieÜbersetzung von `HelloWorld` 209 Klassen mit 1629 Methoden behandelt werden m¨ussen.

Da die Standardklassenbibliotheken so groß sind, erlaubt die Konstruktion des IKFG h¨aufig eine drastische Reduzierung der Gr¨oße des erzeugten Programmes. Da außerdem unerreichbare Methoden sehr fr¨uh w¨ahrend derÜbersetzung gel¨oscht werden, werden alle nachfolgenden Übersetzerphasen, die alle Methoden bearbeiten m¨ussen, beschleunigt. Dies gilt insbesondere f¨ur alle Datenflußanalysatoren, Optimierer und die Codeerzeugung. Diese Beschleunigung ist ein wesentlicher Grund daf¨ur, die Konstruktion des IKFG so genau wie m¨oglich zu machen.

Die entwickelte Analyse basiert auf der gut erforschten Kontrollflußanalyse. Wir wenden diese urspr¨unglich f¨ur funktionale Sprachen entwickelte Technik auf objekt-orientierte Sprachen an. Die vorgestellte Analyse basiert auf der formalen Semantik von `Bahasa`, einer Sprache die einer Teilmenge von Java entspricht. Wir verwenden eine weit verbreitete Java Semantik aus der Literatur als Grundlage unserer Analyse und beweisen bez¨uglich dieser Semantik die Korrektheit unserer Ergebnisse.

Ein Zwischenergebnis der Analyse ist ein System von Constraints ¨uber Mengen, das als gerichteter Graph dargestellt wird. Wir entwickeln einen Bedarf-gesteuerten L¨oser der besondere Eigenschaften des Graphen ausnutzt. Er unterst¨utzt die von uns eingef¨uhrten *Graphfunktoren*, die den Aufrufmechanismus und Zugriff auf Klassenfelder implementieren. Erst der L¨oser bestimmt anhand bereits berechneter Informationen die tats¨achlich aufzurufenden Methoden und zu lesenden oder schreibenden Felder. Hierbei k¨onnen neue Constraints in den Graphen eingef¨ugt werden. Beruhend auf den berechneten Ergebnissen werden Aufrufstellen mit den an ihnen aufrubaren Methoden annotiert. Aus diesen Informationen wird der IKFG konstruiert.

Die Funktoren sind Grundlage eines weiteren Ergebnisses dieser Arbeit. Die Standardklassenbibliothek wird f¨ur dieÜbersetzung aller Programme ben¨otigt, so daß man m¨oglichst viel Information vorberechnen m¨ochte. Der Constraint-Graph mit Funktoren ist modular, d. h. er enth¨alt f¨ur jede Methode einen Teilgraphen, der zu Beginn nicht mit anderen Graphen verbunden ist. Daher k¨onnen wir f¨ur jede Methode eine Beschreibung der Constraints und des Graphen berechnen. Aufgrund der Funktoren k¨onnen diese Beschreibungen vorberechnet werden, da sie unabh¨angig von anderen Methoden und Laufzeit-Informationen sind. Zur Übersetzungszeit werden sie zu dem f¨ur das Benutzerprogramm berechneten Constraint-Graphen hinzugef¨ugt. Dies beschleunigt die Konstruktion des Graphen.

Wir pr¨asentieren eine strukturelle operationale Semantik unserer Sprache und beweisen die Korrektheit unserer Analyse bez¨uglich dieser Semantik. Außerdem demonstrieren wir die Skalierbarkeit durch Anwendung auf realistische Programme. Die Evaluierung wird mit `JoC` durchgef¨uhrt, demÜbersetzer der im `JOSES` Projekt erstellt wurde. Das Projekt wurde im vierten Rahmenwerk als Langzeit-Forschungsprojekt von der Europ¨aischen Gemeinschaft gef¨ordert.

Acknowledgments

The work presented in this thesis has been done at the chair for Compiler Construction of Professor Wilhelm. Unfortunately, there is no total order on the importance of people contributing to this thesis. Therefore, I am going to order them chronological as they appeared in my (or I in their) life. This order will not always be coherent and I apologize to anybody occurring at the wrong place or, even worse, not at all.

The first persons I would like to thank are my parents, Ute and Reiner, and my brother, Michael. They supported me throughout school time, studies and life. They certainly were surprised when they first noticed my interest in computers and electronics back in 1982. However, they never complained when I disappeared for hours, hacking 6502 code or soldering computers.

When I finally came to the University of Saarbr¨ucken, the professor to read the first year course on computer science was Reinhard Wilhelm. I owe him the interest in compiler construction and language theory. He also deserves thanks for letting me work in his group and for the freedom of choosing research topics.

During the first studies I met Florian Martin, whose work on program analyzer generation heavily influenced this work. Using PAG sometimes made life much simpler. During my diploma thesis, Christian Ferdinand allowed me a lot of freedom and certainly encouraged me to stay at university thereafter. At that time I also met Uwe Aßmann, who worked for one of the partners in the project that funded my diploma thesis.

Before starting my Ph.D., Reinhard Wilhelm hired me as a system administrator at the International Meeting and Conference Center, Schloß Dagstuhl. While working at this lovely place, I happened to come to know my wife, Nicole. She deserves special thanks for bearing my research and for encouraging me to continue with my Ph.D. whenever I doubted.

When I returned to the group of Reinhard Wilhelm, it was great fun to work together with Stephan Diehl, Stephan Thesing, Daniel K¨astner and Marc Langenbach. Most of the time, they tolerated my sense of humor – this cannot be overestimated.

The foundations of the work in this thesis have been laid in the JOSES project. Special thanks go to the whole JOSES group, especially Marcel Beemster, Ruben van Royen, Kees van Reeuwijk, Andrzej Bednarzki, Tobias Ritzau, Florian Liekweg, G¨otz Lindenmeyer and Peeter Laud. The integration sessions before reviews where always very intense and productive. Without them, the evaluation of my work would never have been possible. Holger Dewes deserves thanks for

implementing a data-flow version of this work using PAG in his diploma thesis.

Finally, during I wrote down this thesis, J¨org Bauer joined the group. He deserves special thanks for checking and improving the formal part of this thesis. I know he had a hard time convincing me that formalism is not evil.

Reinhard Wilhelm, J¨org Bauer, Stephan Diehl, Stephan Thesing, Marc Langenbach and the `ispell` program proof read more or less preliminary versions of this work. Often, they found subtle errors that probably would have made their way in the final version. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In practice, algorithms such as RTA scale well. The scalability of algorithms such as 0-CFA remains doubtful.

Frank Tip and Jens Palsberg, [TP2000]

It seems to be a popular belief that flow-sensitive construction of the interprocedural control flow graph (ICFG) for object-oriented languages is neither feasible nor scaling. We present a scaling and flow-sensitive approach based on control flow analysis for ICFG construction of statically typed object-oriented programs. Our approach is based on a demand driven solver for constraint based control flow analysis. By introducing functor nodes it allows to pre-analyze library code.

If one believes marketing, object-oriented languages are widely used nowadays in all regions of software development. Repeatedly heard buzz words are *code reuse* and *encapsulation*. However, if one looks into the details, this is not totally true. While *object-oriented design* is quite well accepted, *object-oriented languages* still suffer from minor performance. This is mainly due to two points: the concepts of *dynamic dispatching* and *integrated garbage collection*. While the latter one was introduced to get rid of problems inherent to the memory management of programming languages like `C`, the former one is a property inherently needed for implementing object-oriented languages. However, both concepts hinder an easy application of Java in the world of machine level programming.

As a result, object-oriented languages and especially Java are not yet widespread in software development for embedded systems. Here, developers still prefer languages that are close to hardware, namely `C`. However, Java not only gains more and more influence in education, but also supports the development process with a huge number of class libraries. This results in a growing interest of the embedded systems industry in Java. Figure 1.1 gives the result of an industry study on the languages used in development of embedded systems software [JC1999]. As can be seen the numbers for Java are still rather small compared to `C` and `C++`. However,

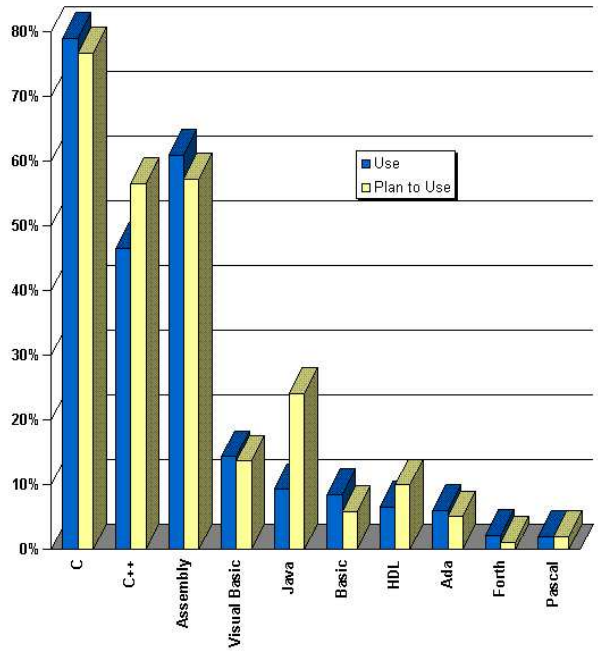


Figure 1.1: Languages used in software development for embedded systems

the most important point is the relative increase for Java. This documents the growing industry interest in Java.

The challenge to be met is that the virtual machine based bytecode approach results in relatively slow performance. A solution is to compile all libraries needed into the program and to generate native code instead of bytecode. As has been seen in the JOSES project [GAF+1999], one needs special optimizations in order to generate programs that run not only fast but also have a reasonable footprint.

Nowadays optimizations to decrease code size and run-time are well understood for imperative languages. There is a large collection of data flow analyses that almost every modern C compiler applies during translation of a program. Usually, these analyses require an interprocedural control flow graph (ICFG). This graph represents the possible program executions at run-time. The construction of the control flow graph is unspectacular for imperative languages without function pointers. For every call statement there is exactly one procedure that is called.

The problem gets harder when we allow the language to support function pointers. Here, the address of a function is taken and passed on in the program. It is then called at some other point. In order to construct an ICFG for this class of languages, the compiler needs to identify addresses of those functions that may be passed on at run-time.

The same problem occurs when translating object-oriented languages like C++ or Java. However, here it is the result of a language feature that is absolutely needed. The programmer defines a

class hierarchy and uses class libraries, where subclasses inherit and/or overwrite methods from super-classes. When an object of a class is created at run-time, the compiler creates a data structure that contains pointers to the implementations of the methods known to the object. If one of the object's methods is to be invoked, the implementation is looked up in this data structure and is then called. The flow of control at a given call site then depends on the types of objects that reach the site at run-time. To create the ICFG under such circumstances, several approaches have been developed. An overview of those approaches will be presented in Chapter 7. Coarsely, they can be classified according to their sensitivity regarding *data flow* and *call contexts*. Those approaches that are totally insensitive are widely spread as they are easy to implement, fast and already provide acceptable results. Nevertheless, related to sensitive approaches the insensitive ones of course can only compute less precise results.

Because of their higher complexity, flow- and/or context-sensitive techniques are usually said to be unusable in practical compilers.

1.1 My Thesis

Our contribution is a demand driven solver for constraint based control flow analysis. To the best of our knowledge this is the first implemented flow-sensitive and scalable analysis for call graph construction for object-oriented languages. By introducing functor nodes we can pre-analyze frequently used library methods and thus speed up analysis. To prove this thesis we develop the analysis used and the solver and prove their correctness. We give algorithms to implement the solver. To demonstrate usefulness and scalability we apply the analysis to benchmarks.

The foundations of this work were laid in the JOSES[1] project [GAF+1999]. The target of the project was to extend the compiler generation system CoSy [AAvS1994] with the techniques needed to support object-oriented languages. The targeted hardware was embedded systems, that usually only provide very few memory resources. In this environment one inevitably needs techniques like those developed in this thesis for two reasons. First, they allow to identify many procedures from the class libraries that will never be called and thus may be deleted before translating the program. Second, these techniques improve the quality of the ICFG generated. Here *quality* is measured by the reduction of the number of superfluous call edges. That is, a better ICFG contains fewer edges.

This is of utmost importance for the compilation process as the time needed by static analyzers of course depends mostly on two things—the number of procedures to visit and the number of call edges. Additionally, the analysis result may be diluted if too many procedures that cannot be called need to be analyzed.

[1] Java and CoSy Technology for Embedded Systems, Esprit LTR project #28198.

1.2 JoC- the JOSES Compiler

During the JOSES project, JoC was developed, a compiler from Java to native code. We will now give a short overview of its structure.

JoC has been implemented in the CoSy framework [AAvS1994] from ACE Associated Compiler Experts b.v. It is an integrated compiler generation system that provides a development environment, an intermediate representation, and analysis and optimization engines that operate on the intermediate representation (IR). Originally, CoSy has been developed to support imperative languages like Fortran and C. In the JOSES project it has been extended to also support object-oriented languages.

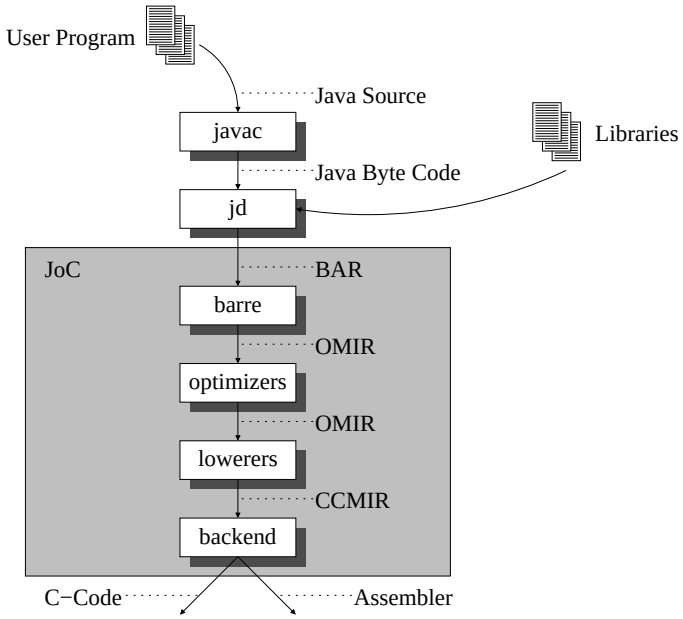


Figure 1.2: The structure of JoC

As a matter of fact, the structure of JoC is mainly imposed by the general structure of CoSy compilers. Figure 1.2 gives an overview of the compiler phases. In order to compile Java source code some preprocessing is applied. First the Java program is translated to bytecode with an arbitrary Java compiler. The advantage of starting the actual compilation chain from bytecode instead of Java is twofold. First, other source languages that can be translated to bytecode may be handled. Second, many complex structures in the source code have been broken down while translating them. Thus, the resulting code is in a format that is better manageable. The bytecode is then translated using jd, a tool specified based on *natural semantics* with RML [Pet1994, Pet1999]. This phase actually unstacks the bytecode and generates an intermediate format named BAR that is closely related both to Java and the intermediate representation used in JoC. It is read into the compiler by barre, the first phase of JoC. Several phases follow that work on the object-oriented intermediate representation OMIR. These include special optimizations including

the one presented in this thesis. At some point in the compiler the object-oriented extensions are lowered away. After this step, the intermediate representation is in the *usual* intermediate format of CoSy compilers, `CCMIR`. What follows are standard optimizations, an outlet for C code and the backend.

1.3 Structure of the Dissertation

The next chapter will provide some theoretical background. The first part gives a brief overview on program analysis. It will introduce different approaches to static analysis and their main properties. Furthermore. some major results and fundamental theorems are presented. The second part presents rule induction. Chapter 3 gives a short introduction into object-oriented languages. Then `Bahasa` is introduced, the language analyzed in this thesis. `Bahasa` is a subset of Java. Additionally, in this chapter an operational semantics for `Bahasa` is introduced.

In Chapter 4 the control flow analysis used in our approach is specified and a proof of its semantic correctness is given. Furthermore, the rules for constraint generation and functors are presented. Chapter 5 is dedicated to the demand driven solver and the algorithms implementing it. Some implementation details are given.

The language introduced in Chapter 3 does not support exceptions, an essential feature of object-oriented languages. These are added to the language and the analysis in Chapter 6.

Chapter 7 presents an overview of related work and compares it to the approach in this thesis.

This will be followed by the evaluation of our practical experiments in Chapter 8. This chapter also presents the results of practical applications.

To conclude, Chapter 9 sums up the results and gives an outlook on future research. The appendix contains additional material on `Bahasa` and the format of pre-compiled information.

Chapter 2

Theoretical Background

This chapter presents some fundamental theorems and definitions for program analysis and rule induction. The former ones allow us to make certain assumptions when specifying our analysis. This includes the termination of the analysis and the existence of (least) solutions. The section on rule induction gives a brief overview on inductive systems. These systems define why the structural operational semantics presented in the next chapter is meaningful.

2.1 Program Analysis

Program analysis is a standard technique in optimizing compilers to determine run-time properties of a given program without actually executing it. The properties of interest are those that can be computed without executing the program and that will hold for all possible executions of it. Examples are the identification of common subexpressions and reaching definitions or the analysis of the shape of data structures on the heap [SRW1999].

However, some of these properties are undecidable, that is, one will not be able to compute a complete *and* correct solution. As one usually does not want to compute an incorrect solution, program analysis makes a compromise on the side of completeness, but ensures to statically predict safe and computable approximations to the dynamic run-time behavior of a program. In Figure 2.1 the region on the left hand side contains data where the analysis correctly states that the analyzed property holds. The right hand side contains data for which the analysis correctly computes the answer *no*. For the data in the middle the analysis computes a *conservative* answer. This means that for all cases that fulfill the property analyzed this will be found out. For some data the analysis still claims that the property holds, while in fact this is not true. What one demands is that the shaded middle region of imprecise answers is as small as possible.

Introductions to data flow analysis can be found in [WM1995], [NNH1999] and [Hec1977], which was the first to provide an overview of lattice based monotone frameworks. [CC1977] introduce *abstract interpretation*, a general framework for specifying static program analyses.

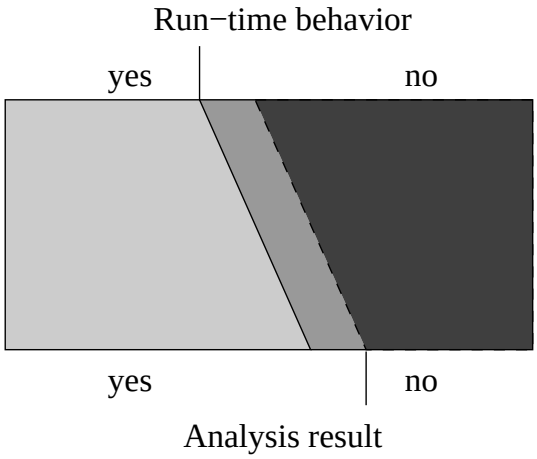


Figure 2.1: Correlation between input data and analysis results

Based on the semantics of the analyzed programming language, it supports correctness proofs of the analysis. Both approaches share the idea to analyze programs using *abstract values* instead of concrete ones. The main motivation to do so is to avoid the uncomputability problem by transforming the analysis into a computable one.

All techniques described have in common that in one way or another they abstract the dependence between data. E. g., data flow analysis uses the control flow graph, control flow analysis labels program points and relates them to each other. We will focus on the intraprocedural case. Section 2.1.3 will describe the common approaches to implement interprocedural analyses.

The following sections will give an overview of the more important techniques. The definitions and theorems are based on [NNH1999], [Mar1999b] and [DP2002].

Object-oriented languages like Java are closely related to imperative languages. Therefore, we will look at those techniques through imperative glasses. For a first overview of the subset of Java we will use, refer to Appendix A. An example program fragment used for illustrating the techniques is given in Figure 2.2. The information we want to compute is *reaching definitions*, i. e. we want to compute for every using occurrence of a variable the definitions that reach it. The table on the right gives the line number of reaching definitions for using occurrences of a variable.

```
x = 1;          //1
if (x>10)
      y = x;    //2
else
      y = 0;    //3
z = y;          //4
```

	x	y
1	-	-
2	1	-
3	1	-
4	1	2,3

Figure 2.2: An example program and its reaching definitions

2.1.1 Data Flow Analysis

Data flow analysis is based on the idea that the program can be modeled as a control flow graph (CFG), the nodes being elementary blocks and the edges modeling the possible flow of control between such blocks. In constructing this graph the analysis can choose between different levels of granularity regarding the nodes – usually the two choices at hand are on the fine side the single statements and on the coarse side basic blocks. The decision will mainly be influenced by the semantics of the language of input programs and the analysis that shall be performed. For our example analysis we map every single statement into its own node in the control flow graph.

The analysis itself will work on the graph and will use it to propagate abstract values from a domain D through the program. The result of the analysis is a labeling of the CFG with values from D that have been computed as being valid before and after the nodes. The values are computed by means of the so called *transfer functions* $f : D \to D$ which model how the abstract values shall be transformed. These functions are associated with the edges in the CFG.

Definition 2.1 *(Control Flow Graph)*
A **control flow graph (CFG)** *is a quadruple $G = (N, E, s, e)$ with a finite set N of nodes, a set $E \subseteq N \times N$ of edges, an entry node $s \in N$, and an exit node $e \in N$.*

Figure 2.3 gives the control flow graph of our example program. Paths between two nodes in a CFG are described as sequences of edges according to the following definition. This will allow an easier treatment of the semantics of execution paths of the program.

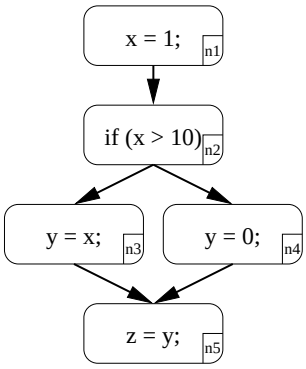


Figure 2.3: The control flow graph for the example fragment

Definition 2.2 *(Path)*
Let $G = (N, E, s, e)$ be a CFG, $n_0, n_k \in N$. A **path** *π from node n_0 to node n_k is a sequence of edges, starting at n_0 and ending at n_k with $\pi = (n_0, n_1), (n_1, n_2), \cdots, (n_{k-1}, n_k)$ where $(n_i, n_{i+1}) \in E, 0 \leq i \leq k-1$.*

The abstract values propagated through the CFG by the analysis are required to form a *complete lattice*. This is needed in order for some of the following theorems to hold.

Definition 2.3 *(Partial Order)*
Let A be a set, $\leq \subseteq A \times A$ a binary relation. $(A, \leq)$ is called a **partial order** *iff for all $a, b, c \in A$:*

$$x \leq x$$
$$x \leq y \wedge y \leq x \Rightarrow x = y$$
$$x \leq y \wedge y \leq z \Rightarrow x \leq z$$

Definition 2.4 *(Complete Lattice, $\bot$, $\top$)*
Let $D = (A, \sqsubseteq)$ be a partially ordered set. D is called a **complete lattice***, iff every subset of A has a greatest lower bound $\sqcap$ and a least upper bound $\sqcup$. The elements $\bot := \sqcap A$ and $\top := \sqcup A$ of D are called bottom respectively top element of D. For the greatest lower bound we write $a \sqcap b$ instead of $\sqcap\{a, b\}$, respectively $a \sqcup b$ instead of $\sqcup\{a, b\}$.*

An important example is the *powerset lattice* $D_{\mathcal{P}} = (\mathcal{P}(A), \subseteq)$ where A is a set. Here, we have $\sqcup \equiv \cup$ and $\sqcap \equiv \cap$. This lattice will be used to abstract concrete data in our analysis in Chapter 4.

Additionally, one usually wants to speak about chains of elements of partially ordered sets.

Definition 2.5 *(Ascending,Descending Chain,Stabilizing Chain)*
Let $D = (A, \sqsubseteq)$ be a partially ordered set, $x_i \in D$ and $I \subseteq \mathbb{N}$. A (possibly infinite) **ascending chain** *$(x_i)_{i \in I}$ is a sequence $x_0, x_1, \cdots$ such that $\forall j \in I : x_j \sqsubseteq x_{j+1}$. A chain $(x_i)_{i \in I}$ is called* **strictly ascending***, iff $\forall j : x_j \neq x_{j+1}$. A chain $(x_i)_{i \in I}$ eventually* **stabilizes***, iff $\exists j \in I : \forall n \geq j : x_j = x_n$.*
Descending chains *are defined analogously.*

Chains will be used to compute abstractions in analyses. Namely, $a \sqsubseteq b$ means that a is more abstract than b. That is, b includes a.

Sometimes we will want to check that a certain set is not empty. This is a property of the following structure.

Definition 2.6 *(Moore Family)*
Let $L = (A, \sqsubseteq)$ be a complete lattice. $M \subseteq L$ is called a **Moore family** *if M is closed under greatest lower bounds.*

A Moore family cannot be empty, because the greatest lower bound of the empty set is the whole lattice and thus must be included in M.

In order to compute an abstraction of the possible run-time values of a program the analysis needs to abstract from the concrete semantics of a programming language. As already mentioned this is performed by means of transfer functions *tf* $: E \to D \to D$ that associate every edge e in the CFG with its abstract semantics. These functions need to have certain properties, too:

Definition 2.7 *(monotone, distributive functions)*
Let $D = (A, \sqsubseteq)$ be a partially ordered set, $f : D \to D$. f is called **monotone***, iff $\forall d, d' \in D : d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$. f is called* **distributive***, iff $\forall d, d' \in D : f(d \sqcup d') = f(d) \sqcup f(d')$.*

Using the transfer functions the semantics of a path in the CFG can be defined. This semantics resembles the semantics of a program execution along that path without the last node.

Definition 2.8 *(Abstract Path Semantics)*
Let $G = (N, E, s, e)$ be a CFG, $D = (A, \sqsubseteq)$ a partially ordered set, $tf : E \to D \to D$ a transfer function and $\pi = e_1, \cdots, e_k$ a path in G. The **abstract path semantics** *$[\pi]_{tf}$ of π is defined as the composition of the transfer function on the edges of π:*

$$\begin{array}{rcl} [\epsilon]_{tf} & = & id_{D \to D} \\ [e_1, \cdots, e_n]_{\mathrm{tf}} & = & [e_2, \cdots, e_n]_{\mathrm{tf}} \circ tf(e_1) \end{array}$$

The solution of a data flow analysis for a node n in G is the least upper bound of the semantics of all paths from entry node s to n, applied to an initial element from D, that is, the combination of the result of all exections reaching n.

Definition 2.9 *(Meet over all Paths Solution)*
Let $G = (N, E, s, e)$ be a CFG, $D = (A, \sqsubseteq)$ a partially ordered set, $\iota \in A$ an initial element and $tf : E \to D \to D$ a transfer function. The **Meet over all Paths solution MOP** *is defined by*

$$MOP(n) = \bigsqcup \{ [\pi]_{tf}(\iota) \mid \pi \text{ is a path from } s \text{ to } n \}$$

Obviously, *MOP* computes for a node n the abstract value directly before the execution of the statement represented by n. By combining the computed values with $\sqcup$ the computed solution is valid for all possible executions of the program represented by G.

However, the *MOP* solution is not always computable, e. g. in the presence of loops one obviously cannot compute the meet over all possible paths. This is why one computes the *minimal fixed point* solution *MFP*. This solution will be computable if three preconditions are fulfilled:

- every ascending chain of the lattice D must eventually stabilize,
- the transfer function $tf(e)$ must be monotone for all $e \in E$, and
- the $tf(e)$ must be computable.

If this is the case, the following definition is constructive in that it describes how to solve the equation system starting from an initial element.

Definition 2.10 *(Minimal Fixpoint Solution)*
Let $G = (N, E, s, e)$ be a CFG, $D = (A, \sqsubseteq)$ a partially ordered set, $\iota \in A$ an initial element and $tf : E \to D \to D$ a transfer function. The **Minimal Fixpoint Solution MFP** *is defined by*

$$\begin{aligned} MFP(s) &= \iota \\ MFP(n) &= \bigsqcup\{tf(e)(MFP(n'))|e = (n', n) \in E\} \text{ for } s \neq n \end{aligned}$$

An important insight for data flow analysis is the so called *coincidence theorem* [KU1977, KS1992]:

Theorem 2.1 *(Coincidence Theorem)*
Let $G = (N, E, s, e)$ be a CFG, $D = (A, \sqsubseteq)$ a partially ordered set, $\iota \in A$ an initial element and $tf : E \to D \to D$ a transfer function.
If $tf(e)$ is monotone for all $e \in E$, and all ascending chains in D eventually stabilize it holds that

$$\forall n \in N : MOP(n) \sqsubseteq MFP(n).$$

If the $tf(e)$ are even distributive, then

$$\forall n \in N : MOP(n) = MFP(n).$$

Thus the *MFP* solution is a safe approximation of the *MOP* solution. Its computation is based on solving the recursive equation system from Definition 2.10. The existence of a solution for this system (the least fixed point) is ensured by *Tarski's Fixed Point Theorem*.

Theorem 2.2 *(Tarski's Fixed Point Theorem)*
Let $(A, \sqsubseteq)$ be a complete lattice. If $f : A \to A$ is a monotone function, then the set of all fixed points of f $FP(f) = \{x|f(x) = x, x \in A\}$ is nonempty and forms a complete lattice when ordered by $\sqsubseteq$.

Kleene's Fixed Point Theorem shows how to iteratively compute the least fixed point of an equation system.

Theorem 2.3 *(Kleene's Fixed Point Theorem)*
Let $(A, \sqsubseteq)$ be a complete lattice where all chains eventually stabilize. If $f : A \to A$ is a monotone function, then there exists a k such that $f^k(\bot) = f^{k+1}(\bot)$ and $f^k(\bot)$ is the least fixed point of f.

Kleene's theorem ensures that the computation will terminate and that the result will be the least fixed point of f. There are several well known techniques to compute fixed points [Mar1999a, FS1999]. The most widely used are based on worklist algorithms.

```
if (x>10)                  //1
      a[1] = new A()[2];   //2
else
      a[3] = new B()[4];   //3
a[5].method(a[6]);         //4
```

Figure 2.4: An object-oriented example program

2.1.2 Control Flow Analysis

Data flow analysis as presented in the previous section allows for the propagation of data through a program. This propagation occurs along the edges of the control flow graph. At every node it is checked whether any information from the data propagated is needed or needs to be changed. Often, however, it is expected to be more efficient to propagate data directly from creation to use points.

Another situation where the control flow graph turns out to be unsuitable is when the successors or predecessors of program points cannot be identified as easily as for imperative languages without function pointers. For imperative languages with function pointers, object-oriented languages, and especially functional languages one usually needs a more powerful technique. If we replace the example fragment by that given in Figure 2.4, an interprocedural analysis will need to be able to identify the implementations of `method` that may be called in line 4.

Control flow analysis (CFA) has been developed in the world of functional languages for the computation of a set of functions that may be called by an expression [Shi1991, NNH1999]. It works by labeling all program points that may contribute to the result of the analysis with labels from $\mathbb{N}$. In the example from Figure 2.4 these labels are shown as superscripts.

The result of a control flow analysis is a pair $(\widehat{\chi}, \widehat{H})$ where

- $\widehat{\chi}$ associates abstract values with program points
- $\widehat{H}$ associates abstract values with variables

This dual to the association of abstract values with nodes in the control flow graph in data flow analysis.

$\widehat{\chi}$ also is called the *abstract cache*, $\widehat{H}$ the *abstract environment*. Assuming that the analysis works on abstract values from a domain $\widehat{\mathbf{Values}}$ the result of a CFA may be formalized as

abstract values	$\widehat{\mathbf{Val}}$	=	$\mathcal{P}(\widehat{\mathbf{Values}})$
abstract environments	$\widehat{\mathbf{Heap}}$	=	$\mathbf{Var} \to \widehat{\mathbf{Val}}$
abstract caches	$\widehat{\mathbf{Cache}}$	=	$\mathbb{N} \to \widehat{\mathbf{Val}}$

The CFA generates a set of equations or constraints. These reflect the definition-use chains in the

program and will allow to compute the abstract environments and caches. Based on the semantics of the analyzed language one may then proof this result to be correct.

In the setting of functional languages an abstract value $\widehat{v} \in \widehat{\mathbf{Val}}$ associated with a label l is a set of functions – namely those terms that may be called when the expression labeled l will be evaluated at run-time. In the object-oriented setting, which we will investigate in Chapter 4, $\widehat{\mathbf{Values}}$ will be the set of classes defined in the program being analyzed. The labeled points are connected, e. g., by set-based constraints in order to model data flow.

As [NNH1999] points out, control flow analysis is quite similar to definition-use chains for imperative languages. Both techniques trace data from points of definition to points of use. When applying CFA to functional languages this means that we trace *function creations* to *function applications*. When computing definition-use chains, definitions are assignments of values to variables and use points are points where these values are used.

When defining a control flow analysis one often first constructs an abstract CFA that will allow to determine whether a pair $(\widehat{\chi}, \widehat{H})$ is an acceptable solution to a given program or not. Acceptability is checked by means of the relation $\models \subseteq (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Heap}} \times \mathbf{Exp})$. This relation is defined along the rules of the semantics of the language analyzed. One possibility to compute a solution for the analysis is to generate constraints from the program and to solve these. This approach will be handled in more detail in Chapters 4 and 5.

Just like data flow analyses, control flow analyses can be distinguished regarding context sensitivity. The simplest form is the so called 0-CFA. In contrast to a DFA, the 0-CFA already handles interprocedural data flow. The labels of the arguments to a method call are connected to the labels of the parameters of the called method. Equally, the label of the return statement of a method is connected to the label of the expression the returned value is assigned to. A more detailed view will be given in Chapter 4.

It is instructive to apply the terminology introduced in Section 2.1.1 to control flow analysis. Labeled program points are equivalent to nodes in the control flow graph. Definition-use chains are modeled by edges, that is the graph contains a path between a node representing a definition and a node representing a use. The lattice used for analysis is $(\mathcal{P}(\widehat{\mathbf{Values}}), \subseteq)$, the powerset of abstract data ordered by the subset relation. Using this analogy we can apply the theorems given above and conclude that there exist fixed points and that we can compute them.

2.1.3 Interprocedural Analysis

Up to now we have only taken *intraprocedural* analysis into account. However, most programs consist of several procedures. This is especially true if library code is analyzed together with the user program. Then one would like to analyze data flow across procedure boundaries. For data flow analysis there exist several solutions to this problem, one of them will shortly be presented following [Mar1999b]. Additionally we will sketch the interprocedural form of control flow analysis, namely k-CFA.

Call String Approach

The approaches for handling of interprocedural data flow analysis include inlining, invalidating, effect calculation, call strings and static call graphs. For an exhaustive explanation see, e. g., [Mar1999b]. We will focus on the call string approach.

The data structure used to represent the interprocedural control flow graph is the *super graph*. Additionally to *normal* edges it contains edges between a call and the method called.

Definition 2.11 *(Super Graph)*
Let P be a program consisting of procedures p_0,...,p_n with control flow graphs G_0,...,G_n, $G_i = (N_i, E_i, s_i, e_i)$. $G^\star = (N^\star, E^\star, s^\star, e^\star)$ is the **super graph** *of P. We have $e^\star = e_0$, $s^\star = s_0$ and*

- $N^\star = \bigcup_{0 \leq i \leq n} N_i'$. *In N_i' each node $n \in N_i$ representing a call statement is replaced by a call node n_c and a return node n_r.*
- $E^\star = \bigcup_{0 \leq i \leq n} E_i \cup E_{\text{call}}$. *In E_{call} there are call edges from each call node n_c to the s_j of methods $\{p_j\}$ callable at n_c and return edges from e_j to n_r. Additionally there is an edge from n_c to n_r.*

The call string approach works by distinguishing calls to the same method by their path through the dynamic call tree. This path can be imagined to model the call stack as present, e. g., in the `C` run-time system. In order to define the call string, all call statements are numbered. Whenever a call is executed, the number of the statement is appended to the call string. When a method returns, the last number in the call string is deleted. The problem with this approach is, that there exist potentially unbounded call sequences in the program. Thus the associated call strings may have infinite length. In order to handle this, the idea is to limit the call string to only contain k numbers, that is the last k call statements executed. The bigger the value of k is chosen, the more contexts can be distinguished. For each method called and each call string, the data flow analysis stores one abstract value for that method.

Returning from a procedure call is problematic. Whenever the length of the call string has reached its maximum length and the kth method m returns, all possible predecessors of m must be assumed to have called the method. This may result in diluted analysis results.

On the one side, choosing a bigger k results in more detailed results. On the other side, also the complexity is increased, as the length of abstract paths grows.

k-CFA

We have already seen that 0-CFA simply connects labels of arguments with labels of parameters. Thus they cannot distinguish two calls of the same method. The idea applied to overcome this problem is exactly the same as for data flow analysis. Also here call statements are labeled and

are used to construct a string of the k last calls executed. Using this string, the abstract domains are extended to distinguish labels and variables occurring in different call contexts.

As with the call string approach for data flow analysis, the complexity of the analysis increases with increasing values for k. For $k > 0$ its complexity is exponential as given in [JW1995].

2.2 Rule Induction

The next chapter will specify the semantics of the language analyzed in this thesis. We will do so by means of a structural operational semantics [Plo1981], that is, we will specify rules with premises and conclusions. By simply specifying the rules, however, it is not at all clear that they have any meaning. This section will provide some fundamental insight. Namely, that the rules we will specify are just an interpretation of an *inductive system*. The presentation follows [Die1996] with the definitions based on [Acz1977, dS1990, dS1992].

2.2.1 Inductive Systems

We start by taking an arbitrary set and defining how to build rules from this set.

Definition 2.12 *(Inductive System, Rules, Axioms)*
Let U be a set. An **inductive rule** *is a pair (P, c) where $P \subseteq U$ and $c \in U$. We call P the premises and c the conclusion of the rule. A rule with $P = \emptyset$ is called an* **axiom**.
An **inductive system** *ϕ is a set of inductive rules. ϕ defines a subset of U. An inductive system is called* finite *if the preconditions of all rules are finite.*

Definition 2.13 *(ϕ-Closed)*
*$A \subseteq U$ is ϕ-***closed** *iff for all $(P, c) \in \phi$ we have that $P \subseteq A \Rightarrow c \in A$.*

Now we assume an arbitrary fixed inductive system and describe the set it defines.

Definition 2.14 *(Inductively Defined Set)*
$\mathcal{I}(\phi) = \bigcap\{A | A$ is ϕ-closed$\}$ is the set **inductively defined** *by ϕ.*

Theorem 2.4 *$\mathcal{I}(\phi)$ is ϕ-closed.*

Proof Let $P \subseteq \mathcal{I}(\phi)$. From Definition 2.14 we get that for every ϕ-closed set A we have $P \subseteq A$. With Definition 2.13 this yields $(P, c) \in \phi \Rightarrow c \in A$. Finally, we get from Definition 2.14 that $c \in \mathcal{I}(\phi)$. This completes the proof. ∎

As $\mathcal{I}(\phi)$ is ϕ-closed and is the intersection of all ϕ-closed sets, we get that $\mathcal{I}(\phi)$ is the least ϕ-closed set.

Example 2.1 *The infinite, inductive system* $\phi_{\mathrm{DA}} = \{(\{m,n\},p)|n,m \in \mathbb{N}, p = m * n\} \cup \{(\{\},2),(\{\},3),(\{\},7)\}$ *defines the set* $\mathcal{I}(\phi_{\mathrm{DA}}) = \{2^i * 3^j * 7^k | i,j,k \in \mathbb{N}\}$.

Theorem 2.5 *(Principle of Rule Induction)*
Let U *be a set,* $\pi \subseteq U$ *a predicate over* U*, and* ϕ *an inductive system on* U.

$$(\forall (P,c) \in \phi : (\forall x \in P : x \in \pi) \Rightarrow c \in \pi)$$
$$\Rightarrow \forall a \in \mathcal{I}(\phi) : a \in \pi$$

We want to model our semantics based on the theory of inductive systems. Now we will define what a proof based on inductive rules looks like. Such a proof is based on a sequence of items. Each item is either an axiom or follows by a rule and a subset of items preceding that item in the sequence.

Definition 2.15 *(Finite Length Proof)*
Let ϕ *be an inductive system on a set* U*,* $\langle b_0, \cdots, b_n \rangle$ *a sequence with* $b_i \in U$*.* $\langle b_0, \cdots, b_n \rangle$ *is a* **finite** ϕ **proof of** b *if* $b = b_n$ *and* $\forall m \leq n : \exists B \subseteq \{b_i : i < m\} : (B, b_m) \in \phi$.

Example 2.2 *Continuing the above example,* $6 \in \mathcal{I}(\phi_{\mathrm{DA}})$ *since* $\langle 2,3,6,12 \rangle$*. 2 and 3 are axioms; 6 and 12 are proven by the rule* $(\{m,n\},p)$.

Theorem 2.6 *For every finite inductive system* ϕ *on* U *we have that* $\mathcal{I}(\phi) = \{b | b \in U \wedge$ b *has a finite* ϕ*-proof*$\}$.

The proof can be found in [dS1992].

As seen in the example, it is not clear in a proof which subset of the sequence has been used to imply an item. That is the sequence does hide the structure of the proof. This can be made explicit using ϕ-trees.

Definition 2.16 *(Proof Tree)*
Let ϕ *be a finite inductive system on the set* U*,* $b \in U$*. A* ϕ**-tree** *or* **proof tree** *of* b*, denoted* $PT_\phi(b)$*, is an object* $\frac{PT_\phi(b_1) \ldots PT_\phi(b_n)}{b}$ *where the rule* $(\{b_1, \ldots, b_n\}, b) \in \phi$ *and* $PT_\phi(b_i)$ *is a* ϕ*-tree for* b_i.

Example 2.3 *A proof tree for 42 in* ϕ_{DA} *is* $\frac{\frac{2 \quad 3}{6} \quad 7}{42}$.

Note that there may be several proof trees for the same proof sequence.

2.2.2 Relational Inductive Systems

Up to now we have only spoken of arbitrary sets and rules and have not yet attached any *meaning* to them. We are now going to add this. To do so, assume that we have a signature Σ and a a set X of variables, with $T_\Sigma(X)$ denoting the set of terms over Σ and X.

The set of all ground terms over Σ is denoted by T_Σ. Ground terms are those that do not contain any variables. Let $t \in T_\Sigma(X)$ be a term and $\theta : T_\Sigma(X) \to T_\Sigma$ be a substitution with $\theta(t) \in T_\Sigma$; then $\theta(t)$ is a ground instance of t.

Definition 2.17 *(Relational Inductive System, Rule, Axiom)*
A **relational inductive rule** *is a pair (P, c) where P is a finite subset of $T_\Sigma(X)$ and $c \in T_\Sigma(X)$. If $P = \emptyset$ then (P, c) is called a* **relational inductive axiom**. *A set of relational inductive rules is called a* **relational inductive system**.

Example 2.4 *To continue the above example, we can now define a finite, relational inductive system, which defines the same set as the infinite, inductive system. To emphasize the difference we use capitals for variables: $\phi_{\mathrm{DA}'} = \{(\{M, N\}, mult(M, N)), (\{\}, 2), (\{\}, 3), (\{\}, 7)\}$.*

Definition 2.18 *(Evaluation of Functions)*
Assume that the names in Σ can be divided into function names F and constructor names C with $\Sigma = F \cup C$ and $F \cap C = \emptyset$. Additionally, let $\alpha : F \to \mathbb{N}$ map each function to its arity, $m = max(\{\alpha(f) | f \in F\})$ and let $\sigma : F \to ((T_C^0 \cup \cdots \cup T_C^m) \to (T_C \cup \{\bot\}))$ be an interpretation of the function names. The **evaluation** *$\eta_\sigma(t)$ of a ground term t by an interpretation σ is defined as:*
Let $t = f(t_1, \ldots, t_n)$. There are two cases:

$$
\begin{array}{lll}
1. f \in F & \eta_\sigma(t) = \sigma(f)(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n)) & \text{if } \forall i : \eta_\sigma(t_i) \neq \bot \\
 & \eta_\sigma(t) = \bot & \text{otherwise} \\
2. f \in C & \eta_\sigma(t) = f(\eta_\sigma(t_1), \ldots, \eta_\sigma(t_n)) & \text{if } \forall i : \eta_\sigma(t_i) \neq \bot \\
 & \eta_\sigma(t) = \bot & \text{otherwise}
\end{array}
$$

η_σ naturally extends to sets and tuples.

Example 2.5 *The signature in our example is $\Sigma = \mathbb{N} \cup \{mult\}$, where the interpretation σ maps $mult$ to the multiplication of natural numbers.*

Definition 2.19 *(Derived Inductive Systems)*
Let ϕ be a relational inductive system. The **inductive system $\bar{\phi}$ derived from** *ϕ by an interpretation σ is the set of all rules $\eta_\sigma(\theta((\{p_1, \ldots, p_n\}, c)))$ such that $(\{p_1, \ldots, p_n\}, c) \in \phi$, θ is a substitution and $\theta(p_1), \ldots, \theta(p_n), \theta(c)$ are ground instances.*

Example 2.6 *The derived inductive system for our example is* $\bar{\phi}_{\mathrm{DA}'} = \{(\{m, n\}, p)|m, n \in \mathbb{N} \wedge p = m * n\} \cup \{(\{\}, 2), (\{\}, 3), (\{\}, 7)\}$. *Obviously, this is equal to the inductive system* ϕ_{DA}. *As a consequence, the set defined by the relational inductive system is the same as that defined by the derived inductive system, namely* $\mathcal{I}(\bar{\phi}_{\mathrm{DA}'}) = \mathcal{I}(\phi_{\mathrm{DA}}) = \{2^i * 3^j * 7^k|i, j, k \in \mathbb{N}\}$.

Having presented this excerpt of the theory of inductive systems, the question remains how that helps us. Actually, what we are going to use as semantics are just rules on a set of terms. Inductive systems as base theory allow us to assume that with the proper interpretation of functions and constants we end up with a set derived from this system. The proof trees will describe evaluations of programs. Example 3.6 will exemplarily show such an execution. The set defined by the system contains all possible end configurations of evaluations.

Chapter 3

The Language

The contents of a language is revealed at best by those words that cannot be translated.

Marie von Ebner-Eschenbach

This chapter introduces the object-oriented language `Bahasa` that will be analyzed in the rest of this thesis. After a short introduction to object-oriented languages we give the abstract syntax of `Bahasa`. A program will conceptually be split into two parts – namely the methods and statements on the one side and a type system on the other. The type system will allow us to investigate the class hierarchy of programs and the types of methods and variables. As expected, the statements will be used for describing evaluation of `Bahasa` programs. This is done by means of a small step structural operational semantics [Plo1981] for the language. The semantics will allow us to prove the correctness of the analysis presented in the next chapter. The chapter concludes with the formal execution of a small example program.

We start with a very short introduction to object-oriented languages, exemplified by Java. This section shall just ensure that the more common vocabulary of object-oriented languages is at hand.

3.1 Object-Oriented Languages

Imperative programming languages support the programmer with the concept of procedures for modularization. In object-oriented languages the main concept is the abstract data type, that is, the encapsulation of data and functionality. Of course, many concepts can be found in variants in different language families. We are going to mention those similarities whenever appropriate. Note that we will not further introduce the syntax of Java.

As the name says, in object-oriented languages the main entities to work on are *objects*. They

can be characterized by a set of variables and methods that access those variables. The variables are said to reflect the *structure* of an object, the methods to reflect its *behavior*.

Example 3.1 *(Objects)*
Suppose we would like to model candidates for an election. Of course each candidate object should have a name, a party it represents, and the number of ballots that have voted for the candidate represented by it. The interface should contain some functionality to get the name of a candidate, her party, and the number of votes she got. Additionally one would expect a way to increase the number of votes, which should be called whenever a ballot voting for the candidate has been found.

As can be seen, objects representing similar entities often have a lot of properties in common. Furthermore, they usually are expected to support a somehow similar interface. For this reason object-oriented languages introduce the concept of *classes*. A class describes objects having the same type, that is, the same structure and the same interface. That's how the user may introduce new types into the language used. The concept of *classes* can be compared with *records* in imperative languages.

By means of a special function `new` an object adhering to a class may be created at run time. Furthermore, *constructors* may be defined, that initialize the fields of an object when it is created. Constructors do have the same name as the class they are defined in and syntactically do not have a return type.

In addition to instance functions and methods, that are local to each object, Java also supports *class* variables and methods. These are shared by all instances of a class. They are used like global variables in imperative programming languages. Both class methods and variables are marked by the keyword `static`.

Example 3.2 *(Java source code)*
A Java class definition for the election example would look like

```
class Election {
          String name;
          String party;
          int votes;
          static int year;

          Election() { votes = 0; }
          Election(String CandName,String CandParty) {
              name = CandName; party = CandParty; }

          String getName() { return name; }
          String getParty() { return party; }
          int getVotes() { return votes; }
          void tick() { votes=votes+1; }
}
```

As can be seen, the class defines three instance variables and one class variable. While the year of the election should be the same for all instances of the class, the name, the party and the number of votes of course will be local to each instance, i. e. candidate.
The constructor gets the name of the candidate and her party and stores these values in the instance variables. Furthermore, the number of votes is set to 0*. The interface of the class just defines several methods to read the instance variables and to increase the number of votes.*

Another important feature of object-oriented languages is *inheritance*, which allows to extend the type hierarchy of a language. This concept exists in several variations, each one with its own problems. We are not going to investigate this topic further. Java is a language that supports *single* inheritance, that is every class in Java has a single super class. This is exactly how inheritance works – when declaring a class one may specify a super class. From this the class inherits all methods and fields. Methods may be overwritten by methods with the same argument types. While the class that is extended is called the *parent* or *super* class, the inheriting class is dually called the *child* or *sub* class.

Example 3.3 *(Inheritance)*
Suppose that in addition to an usual election you would also like to be able to decrease the number of votes a candidate has got. That is you basically want the same structure and interface as in the class `Election`*, but additionally a method to decrease the number of votes. Furthermore a ballot should be worth two votes instead of one. This can easily be realized by using inheritence:*

```
class Election2 extends Election {
                Election2(String CandName,String CandParty) {
                    name = CandName; party = CandParty; }

                void tick() { votes=votes + 2; }
                void untick() { votes=votes - 2; }
}
```

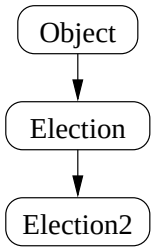
The new class specifies that it extends class `Election` *and thus inherits all variables and functionality from it. It then overwrites the definition of tick and defines a new method untick.*

As already said, inheritance extends the type system of object-oriented languages. A class is treated as a type (often these two are even used as synonyms) and a child class as a sub type. This leads to the so called *class hierarchy* that states the relations between classes, i. e. types. It is the set of class names, partially ordered by the binary *is subclass of* relation.

Object-oriented languages often have a root class, that all classes inherit some basic structure and interface from. In Java this class is `java.lang.Object`.

Example 3.4 *(Class Hierarchy)*

The class hierarchy for the two classes just defined would look like the graph shown on the right hand side. In literature one often finds the arrows pointing from the child to the parent class. Of course there will be many more sub classes to the Object class in a real world program.



For languages like Java, that have a root class, the class hierarchy graph always is a tree.

The class hierarchy (that is type hierarchy) is an important tool when compiling object-oriented languages. A variable `v` has a declared type, say `T`. This type identifies which objects may be stored in `v` at run-time. Namely these are objects of those classes that are descendants of `T` in the class hierarchy graph. E. g., if we have a variable with type `Election`, then we may also store objects of class `Election2` in it. Why is this? As we have seen above, objects of `Election2` share the same structure and interface with objects of `Election`, the parent class. That is, a call `v.tick()` is known to succeed, indepent of the run-time object that is stored in `v`. The only thing to be checked at compile-time is, that only expressions typed with a sub type of `T` are assigned to `v`. Allowing super types would fail, as e. g. objects of class `Election2` support the method *untick* that is unknown in the super classes.

Java also allows to specify *interfaces*. Interfaces do not provide method implementations but just method signatures. Beside extending *one* super class, a class can specify several interfaces it assures to implement. This means that the class must provide an implementation for each of the prototypes in any of the interfaces it implements. Thus the problems combined with multiple inheritance can be avoided. For a detailed discussion of the inheritance problems c. f. [WM1995].

A main difference between imperative and object-oriented languages is the way methods are called. In imperative languages a method call will result in a jump to a certain address at run-time. If the language does not have function pointers this address is even known at compile-time. In object-oriented languages the idea is that an object decides at run-time which method to call. The semantics of the used programming language defines how this decision is to be made. This concept is known as *dynamic dispatch*.

As just explained, using the declared type of a variable it can be checked at compile-time that the run-time objects will support the method. However, it is not known, which class the objects will belong to at run-time. Due to inheritance, different classes may provide different implementations with the same signature. That's why the object needs to decide which implementation to call.

Example 3.5 *(Method dispatch)*
Assume the following code fragment, using the classes defined above.

```
Election cand;
Election2 cand2;
```

```
cand = new Election2("Donald", "Loosers");
cand.tick(); cand.tick();
cand2=(Election2)cand; cand2.untick();
```

Assigning an object of class `Election2` *to a variable with declared type* `Election` *is correct as the object's type is a sub type of the variable's type. The object is then sent the message to invoke the method* `tick`*. As the object is of class* `Election2` *it will invoke the implementation that increases the number of votes by two. The method* `untick` *cannot be called on the variable* `cand`*, as class* `Election` *does not know about it. However, after assigning cand to cand2 we may call* `untick`*, as* `Election2` *implements the method.*

Each dynamically dispatched method has an implicit parameter `this`, that points to the object on that the method has been called. `this` has the declared type of the class that defines the method.

There are many more concepts associated with object-oriented languages. Classes may inherit from several classes, objects may be allowed to change their interface during run-time, languages may support generic data types and many more. However, those concepts described up to now are the essence of the class of languages that Java belongs to. For a more detailed introduction see, e. g., [BH1998].

3.2 `Bahasa` – a Subset of Java

Since several years the object-oriented language to analyze is Java [GJS1996]. Complete Java contains many constructs that do not contribute to the expressiveness of the language. As in the JOSES framework we use bytecode as input and unstack it to Java code, our input language consists of a kind of simple structured subset of the full language. Here, we will define this subset of Java, and name it `Bahasa`. This choice of name is inspired by the fact that the island Java belongs to Indonesia. There the official language spoken is *Bahasa Indonesia*.

The complete syntax of `Bahasa` can be found in Appendix A. Table 3.1 presents a context-free grammar with regular expressions as auxiliary constructs. This allows us to avoid the cumbersome derivations for lists. As can be seen, the language is almost equal to Java. However, it is somehow simplified in order to allow a convenient treatment. It is defined along the lines of and borrowing many notations from [DE1999]. Without loss of generality one may assume that all values are assigned to local variables before being used. Furthermore, all occurences of `this`, that are usually implicit, have been made explicit as in `o.method(o)` instead of `o.method()`. This results in a second change, that is, we need to mark all static methods because they could not be distinguished from dynamic ones (e. g. `static void test(A a)` and `void test()` in class *A* both look equal after making `this` explicit). The marking is done in a type environment Γ that will be defined later. The same holds for static fields as those are shared by all instances created for a class. Finally, we need to mark constructors. Those are identified by having no return type and the same name as the class they are defined in.

Beside this, the most noteable changes from Java to `Bahasa` are

- all variables have unique names
- constructors are called explicitly, i. e., the `new` method just delivers an object where all member fields have been initialized to the initial values of their type
- accesses to instance and class members are only one level indirections
- there is no `super` keyword
- casts to a class *C* are replaced by assignments to a variable with static type *C*
- the language has neither arrays nor exceptions
- expressions are not further specified
- every class beside `java.lang.Object` extends some other class
- interfaces may have only methods but no fields

The first three changes are merely inspired by the framework that the analysis has been implemented in.

The `super` keyword is just a syntactic enhancement – it actually tells the compiler to call the method to invoke on the superclass of the class containing the actual implementation. Thus at compile-time the method to be called can be identified as this decision is independent of the run-time type of an object.

Casts in Java only have two reasons to exist – one of them is to force the compiler to access member fields defined in the class casted to. The other is to tell the type checker that the expression has a more specific type than the type checker might find out. As we trace actual types through the program and thus know a set of correct types for each expression we can get rid of the second type of casts. The first, however, can be simulated by assigning an expression to a variable with the type casted to as declared type.

Arrays would not add any additional problems due to the kind of our analysis (c. f. Chapter 4). We will identify variables with types of objects possibly stored in them. This approach also handles arrays by identifying them with the array class. That is, the analysis can handle arrays, but their addition to the language would blow it up unnecessarily. Exceptions and their addition are handled in Chapter 6. Additionally, we make `java.lang.Object`, the super class of all Java classes, explicit for all classes that do not have another super class. From now on we will refer to this class by `Object`.

Expressions in `Bahasa` are simply constants. The only explicit values are truth values, all other primitive types, e. g., integers and characters, as well as arithmetic or boolean expressions are represented by the constant `c`. This design decision tries to emphasize once more that we are only concerned with classes and not with primitive types.

Program	:=	(*InterfaceDef*)* (*ClassDef*)*
InterfaceDef	:=	`interface InterfaceId` [`extends` (`InterfaceName`)+] {*ReturnType* `MethodId (` (*VarType* `ParamId`)*`);`}
ClassDef	:=	`class ClassId` [`extends ClassName`] [`implements` (`InterfaceName`)+] { *VariableDefinition** *MethodDefinition** }
Modifier	:=	`static` | ϵ
VariableDefinition	:=	*Modifier VarType* `VarId;`
LocVariableDefinition	:=	*VarType* `VarId;`
MethodDefinition	:=	*Modifier ReturnType* `MethodId (`(*VarType* `ParamId`^{l_i})*`)` { *LocVariableDefinition** *Stmts* ; }^{l_r}
Block	:=	{ *Stmts* } | *Stmt* ;
Stmts	:=	*Stmts Stmt* ; | ϵ
Stmt	:=	`if` *Expr* `then` *Block* `else` *Block* | `skip`
	|	*Var* ^{l_1} `=` *Rhs* ^{l_2} | *Call*
	|	`return` [*Variable*^{l}];
Rhs	:=	*Var* | `new ClassName()` | *Call* | *Expr*
Call	:=	*Variable* ^{l_o}`.MethodName(` *Variable* ^{l_i*} `)`
Var	:=	*Variable* | *Member*
Variable	:=	`VarName`
Member	:=	`ClassName`^{l_o}`.MemberName` | *Variable* ^{l_o}`.MemberName`
ReturnType	:=	*Type* | `void` | `cstr`
VarType	:=	*Type* | `InterfaceName`
Type	:=	`ClassName` | `int` | `boolean` | `char`
Expr	:=	`true` | `false` | `c`

Table 3.1: `Bahasa` syntax

Interface fields are deleted due to their semantics in Java. They are *final*, that is they may only be assigned a value at declaration and are then used like constants. We therefore replace their occurrences by the value that is assigned to them.

These changes altogether allow us to make some assumptions about the programs that will be analyzed. Moreover, they help us to keep the language grammar, the typing rules, and the analysis itself quite simple by removing much of the overhead introduced in Java.

The analysis that will be presented in Chapter 4 will need to speak about program points in `Bahasa` programs. Therefore those points are labeled. The labels l are taken from the set of natural numbers. They are assigned to those expressions that may result in an object. Accordingly, parameters and variables are only labeled if they have class type. Similarly, we assign a label to

the body of a method `m` if the return type of the method is a class type. This label will be used to collect all the values possibly returned by the method. Furthermore, the objects that are either sent messages for method invocation or whose fields are accessed are labeled. For reasons of symmetry we also label the class that is used for access to class members. To be more specific, we label all using and defining occurrences of expressions that will result in a class type at run-time. The places that are labeled are given in Table 3.1. Of course, the labels do not change the semantics of the language.

In what follows we will develop a semantics for `Bahasa`. This will be based on ideas from [DE1997, DE1999, NvO1998].

In order to allow for a treatment of execution of `Bahasa` programs we will use a slightly modified version of the language, named $\texttt{Bahasa}_r$(c. f. Table 3.6). Its construction from `Bahasa` is inspired by that of Java_r in [DE1999]. In addition, we introduce constructs from Plotkin's SOS [Plo1981]. The main difference to `Bahasa` is that now there may occur addresses. This is due to the fact that run-time references to objects may occur that are not visible at the *normal* language level. Therefore we introduce a new intermediate expression that represents those references. Additionally we need to model the run-time stack when invoking methods. To do so, similarly to intermediate expressions we introduce two more intermediate statements `bind` and `end`. Just like references these statements do not occur in `Bahasa` programs but only at run-time. Using `bind`, the run-time stack will be encoded into the intermediate programs. `end` will identifiy the end of the statements that belong to a method in order to delete them from the configuration if the method's evaluation is finished.

3.3 A Type System for `Bahasa`

Before handling the run-time model for `Bahasa`, we will establish a type system. Namely, as `Bahasa` is just a subset of Java, one can take whatever one's favorite Java type system is and use that to type `Bahasa` programs. We are actually not concerned with typing whole programs and each and every statement or expression. However, we need to speak about type hierarchy, subclass relations, and method definitions. That's why we have chosen a standard Java type system and have extracted the part that deals with the class structure imposed by a program. Thus from the program *p* a class hierarchy can be deduced. For reasons of convenience we will identify the type of class *C* with its name. For a complete treatment of how to type Java programs compare, e. g., [DE1999].

We start by defining some semantic domains.

Definition 3.1 *(Semantic Domains, Subclass Relation)*
Let p be a `Bahasa` program. We define a number of semantic domains and give typical meta variable we will use:

$e, \epsilon \in$ ***Term*** *(possibly partial) `Bahasa` terms*

$$\frac{p = p' \texttt{ class C extends C}' \texttt{ [implements } \texttt{I}_1\texttt{, } \cdots\texttt{, I}_n \texttt{] \{ } \cdots \texttt{ \} } p''}{\begin{array}{l}\texttt{C} \sqsubseteq_{\texttt{p}} \texttt{C} \\ \texttt{C} \sqsubseteq_{\texttt{p}} \texttt{C}'\end{array}} \qquad \frac{\texttt{C} \sqsubseteq_{\texttt{p}} \texttt{C}' \quad \texttt{C}' \sqsubseteq_{\texttt{p}} \texttt{C}''}{\texttt{C} \sqsubseteq_{\texttt{p}} \texttt{C}''}$$

Table 3.2: Subclasses deduced from program p

$l_1, l_r \in$	$\mathbb{N}$	*a set of labels to be used in programs*
$\texttt{C}, \texttt{C'} \in$	***Classes***	*the set of class names defined in p*
$\texttt{I}, \texttt{J} \in$	***Interfaces***	*the set of interface names defined in p*
	OOTypes	$:=$ ***Classes*** $\cup$ ***Interfaces***
	Types	$:=$ ***OOTypes*** $\cup \{\mathit{int}, \mathtt{bool}, \mathtt{char}\}$
$MT \in$	***MTypes***	$:= \{T_1 \times \cdots T_n \to T_r \mid n \geq 0, 1 \leq i \leq n,$ $T_i \in$ ***Types***$, T_r \in$ ***Types*** $\cup \{void\}\}$
$\texttt{v} \in$	***VNames***	*the set of names of local variables in methods in p*
$\texttt{f} \in$	***FNames***	*the set of field names defined in classes in p*
$\texttt{m} \in$	***MNames***	*the set of method names defined in classes in p*

$\sqsubseteq_p \subseteq$ ***Classes*** $\times$ ***Classes*** *defines the subclass relation imposed by program p. Table 3.2 defines how to deduce this relation from p.*

For the rest of the chapter we assume a partitioning of `Bahasa` programs in the type-related part and the execution-related part. The type environment contains classes, interfaces, methods, and variable definitions. In addition, it stores the class and interface hierarchy. The actual program contains the methods and statements. This partition is performed once for a given program and is fixed during analysis. A class declaration introduces a new class `C` as a subclass of a class `C'`, several fields and methods, and optionally, a list of interfaces implemented by `C`. Field declarations consist of the name of the field and its type, method declarations of their identifier, and their type. Both fields and objects may have the modifier `static`, identifying those members that are shared by all instances of the class.

An interface declaration introduces a new interface `I`, possibly as a sub-interface of a list of other interfaces. Due to our assumptions on the structure of `Bahasa` programs as explained at the beginning of this chapter, components of `I` are methods only.

We will now introduce the type environment, a function that will store relationships between classes and interfaces, fields and types, and methods and types and local variables. The domain of Γ ranges over class and interface names. For each of these, Γ stores

- the super class (ϵ for interfaces)

- the set of implemented or extended interfaces
- a function from field names defined in a class to a pair, containing
  - a flag whether the field is static
  - the type of the field
- a function from a pair consisting of method name and method type to a triple containing
  - a flag whether the method is static
  - a flag whether the method is a constructor
  - a function ranging over local variable names returning their declared types

We will write MT_j for a method type as abbreviation of $(\mathtt{PT}_1 \times \cdots \times \mathtt{PT}_{a_j}) \to \mathtt{RT}_j$.

Definition 3.2 *(Type Environment Γ)*
Let p be a `Bahasa` *program,* ***Static*** $=$ ***Constructor*** $=$ ***Bool****. The* **type environment** Γ *induced by p is a function*

$$\begin{aligned}\Gamma : \textbf{\textit{OOTypes}} \quad \to \quad & (\textbf{\textit{Classes}} \cup \{\epsilon\}) \times \mathcal{P}(\textbf{\textit{Interfaces}}) \times \\ & (\textbf{\textit{FNames}} \to (\textbf{\textit{Static}} \times \textbf{\textit{Types}})) \times \\ & ((\textbf{\textit{MNames}} \times \textbf{\textit{MTypes}}) \to (\textbf{\textit{Static}} \times \textbf{\textit{Constructor}} \times (\textbf{\textit{VNames}} \to \textbf{\textit{Types}})))\end{aligned}$$

Γ *is constructed from p for class and interface declarations in p.*

$p \equiv p$'`class C extends C' implements` $\mathtt{I}_1$`, ...,` $\mathtt{I}_n$ `{`
`fmod`$_1$ $\mathtt{FT}_1$ $\mathtt{f}_1$`; ...` `fmod`$_{m_1}$ $\mathtt{FT}_m$ $\mathtt{f}_m$`;`
`mmod`$_1$ $\mathtt{RT}_1$ $\mathtt{m}_1$ `(`$\mathtt{PT}_{(1,1)}$ $\mathtt{p}_{(1,1)}$ $\cdots$ $\mathtt{PT}_{(1,a_1)}$ $\mathtt{p}_{(1,a_1)}$`) {`
$\mathtt{VT}_{(1,1)}$ $\mathtt{lv}_{(1,1)}$`; ...` $\mathtt{VT}_{(1,v_1)}$ $\mathtt{lv}_{(1,v_1)}$`;`
`Stmts }`
`...`
`mmod`$_l$ $\mathtt{RT}_l$ $\mathtt{m}_l$ `(`$\mathtt{PT}_{(l,1)}$ $\mathtt{p}_{(l,1)}$ $\cdots$ $\mathtt{PT}_{(l,a_l)}$ $\mathtt{p}_{(l,a_l)}$`) {`
$\mathtt{VT}_{(l,1)}$ $\mathtt{lv}_{(l,1)}$`; ...` $\mathtt{VT}_{(l,v_l)}$ $\mathtt{lv}_{(l,v_l)}$`;`
`Stmts }` p"

$$\begin{aligned}\Rightarrow \Gamma(\mathtt{C}) = (\mathtt{C}', \{\mathtt{I}_1, \cdots, \mathtt{I}_n\}, \\ [\mathtt{f}_i \to (stat_i, \mathtt{FT}_i)]_{i=1,\ldots,m}, \\ [(\mathtt{m}_i, MT_i) \to (stat_i, cons_i, [\mathtt{lv}_j \to \mathtt{VT}_j, \mathtt{p}_k \to \mathtt{PT}_k])]_{i=1,\ldots,l;j=1,\ldots,v_i;k=1,\ldots,a_i})\end{aligned}$$

$p \equiv p'$ `interface I extends` $\mathtt{I}_1$`, ...,` $\mathtt{I}_n$ `{`
`mmod`$_1$ $\mathtt{RT}_1$ $\mathtt{m}_1$ `(`$\mathtt{PT}_{(1,1)}$ $\mathtt{p}_{(1,1)}$ $\cdots$ $\mathtt{PT}_{(1,a_1)}$ $\mathtt{p}_{(1,a_1)}$`);`
`...`
`mmod`$_l$ $\mathtt{RT}_l$ $\mathtt{m}_l$ `(`$\mathtt{PT}_{(l,1)}$ $\mathtt{p}_{(l,1)}$ $\cdots$ $\mathtt{PT}_{(l,a_l)}$ $\mathtt{p}_{(l,a_l)}$`); }` p''

$$\Rightarrow \Gamma(\mathtt{I}) = (\ \epsilon, \{\mathtt{I}_1, \cdots, \mathtt{I}_n\}, [], \\ [(\mathtt{m}_j, MT_j) \rightarrow (stat_j, \mathtt{false}, [])]_{j=1,\ldots,l})$$

where

$$stat_j = \begin{cases} \mathtt{true} & \textit{if } \mathtt{fmod}_j \textit{ respectively } \mathtt{mmod}_j \textit{ equals } \mathtt{static} \\ \mathtt{false} & \textit{otherwise} \end{cases}$$

$$cons_j = \begin{cases} \mathtt{true} & \textit{if } \mathtt{RT}_j = \mathtt{cstr} \\ \mathtt{false} & \textit{otherwise} \end{cases}$$

Interfaces are not allowed to define constructors for obvious reasons, so the constructor flag in Γ is always `false` for interface functions. As those functions are also only declarations, they cannot have any local variables.

Since the local variables in methods all have unique names, we may merge the functions mapping those variables to their type.

Definition 3.3 *(Function Γ_V)*
Let p be a Bahasa program and Γ the type environment induced by p.
*The function Γ_V : **VNames** $\rightarrow$ **Types** returns the declared type of a variable.*

$$\Gamma_V(\mathtt{v}) := \begin{cases} f_{\text{local}}(\mathtt{v}) & \textit{if } \exists \mathtt{C} \in \textbf{\textit{Classes}}, \mathtt{m} \in \textbf{\textit{MNames}} : \Gamma(\mathtt{C}) = (\mathtt{C}', M_I, f_f, f_m) \wedge \\ & \quad f(\mathtt{m}) = (b_s, b_c, f_{\text{local}}) \wedge \mathtt{v} \in dom(f_{\text{local}}) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Just like program *p*, the environment induced by *p* also defines a subclass relation $\sqsubseteq_\Gamma$ and a dual relation $\leq_\Gamma$ on interfaces. Furthermore an *implements* relation $:_{imp}$ between classes and interfaces is defined in Table 3.3.

In order to declare variables and methods, one needs to be able to impose restrictions on their possible types. Those restrictions are given in Table 3.4. Their main reason is to ensure that a class occurring in a method or variable declaration is really defined in the program.

Definition 3.4 *(Functions ArgTypes, ResType)*
Let p be a Bahasa program and Γ the environment derived from p.

*The functions ArgTypes : **MTypes** $\rightarrow \bigcup_{i \geq 1}$ **Types**^{i} and ResType : **MTypes** $\rightarrow$ (**Types** $\cup \{\mathtt{void}\}$) return the type of the arguments respectively the return type of a given method type, i. e.*

$$\frac{\Gamma(\mathtt{C}) = (\mathtt{C}', \{\mathtt{I}_0, \cdots, \mathtt{I}_n\}, f_{fields}, f_{methods})}{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C},\ \mathtt{C} \sqsubseteq_\Gamma \mathtt{C'},\ \mathtt{C} :_{imp} \mathtt{I}_i,\ 1 \le i \le n}$$

$$\frac{\Gamma(\mathtt{I}) = (\epsilon, \{\mathtt{I}_0, \cdots, \mathtt{I}_n\}, [], f_{methods})}{\mathtt{I} \le_\Gamma \mathtt{I},\ \mathtt{I} \le_\Gamma \mathtt{I}_i,\ 1 \le i \le n}$$

$$\frac{}{\mathtt{Object} \sqsubseteq_\Gamma \mathtt{Object}} \qquad \frac{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}' \quad \mathtt{C}' \sqsubseteq_\Gamma \mathtt{C}''}{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}''} \qquad \frac{\mathtt{I} \le_\Gamma \mathtt{I}' \quad \mathtt{I}' \le_\Gamma \mathtt{I}''}{\mathtt{I} \le_\Gamma \mathtt{I}''}$$

Table 3.3: Relations defined by environments

$$\frac{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}}{\mathtt{C} \Diamond_{\mathtt{VarType}}} \qquad \frac{\mathtt{I} \le_\Gamma \mathtt{I}}{\mathtt{I} \Diamond_{\mathtt{VarType}}} \qquad \frac{}{\begin{array}{c}\mathtt{int} \Diamond_{\mathtt{VarType}} \\ \mathtt{char} \Diamond_{\mathtt{VarType}} \\ \mathtt{bool} \Diamond_{\mathtt{VarType}}\end{array}}$$

$$\frac{\begin{array}{c}\mathtt{T} \Diamond_{\mathtt{VarType}} \text{ or } \mathtt{T} = \mathtt{void} \\ \mathtt{T}_i \Diamond_{\mathtt{VarType}},\ 1 \le i \le n\end{array}}{\begin{array}{c}\mathtt{T}_1 \times \cdots \times \mathtt{T}_n \Diamond_{\mathtt{ArgType}} \\ \mathtt{T}_1 \times \cdots \times \mathtt{T}_n \to \mathtt{T} \Diamond_{\mathtt{MethType}}\end{array}}$$

Table 3.4: Variable and method types

$$\begin{aligned} \mathit{ArgTypes}(\mathtt{T}_1 \times \cdots \times \mathtt{T}_n \to \mathtt{T}_r) &= \mathtt{T}_1 \times \cdots \times \mathtt{T}_n \\ \mathit{ResType}(\mathtt{T}_1 \times \cdots \times \mathtt{T}_n \to \mathtt{T}_r) &= \mathtt{T}_r \end{aligned}$$

Finally, we need a widening relation $\le_w \subseteq$ **OOTypes** $\times$ **OOTypes** between variable types. This relation will allow us to check assignments between variables and expressions with different types. Such an assignment is possible if the type of the expression is a subtype of the type of the variable. Table 3.5 shows the rules for widening of class and interface types. An assignment between a variable with type *T* and an expression with type $\mathtt{T}'$ is allowable if $\mathtt{T}' \le_w \mathtt{T}$. In addition to the rules given in the table, [GJS1996] describe how to widen primitive types. As our main objective are class types we skip these.

$$\frac{\mathtt{C} \Diamond \mathtt{VarType}}{\mathtt{C} \leq_w \mathtt{C}} \qquad \frac{\mathtt{C} \leq_w \mathtt{Object}}{\mathtt{nil} \leq_w \mathtt{C}} \qquad \frac{\mathtt{C} \leq_w \mathtt{C}}{\mathtt{C} \leq_w \mathtt{Object}}$$

$$\frac{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}'}{\mathtt{C} \leq_w \mathtt{C}'} \qquad \frac{\mathtt{I} \leq_\Gamma \mathtt{I}'}{\mathtt{I} \leq_w \mathtt{I}'} \qquad \frac{\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}', \mathtt{C}' :_{imp} \mathtt{I}, \mathtt{I} \leq_\Gamma \mathtt{I}'}{\mathtt{C} \leq_w \mathtt{I}'}$$

Table 3.5: Widening of class and interface types

Next, we specify when an environment is *well-formed*. Later on, this will allow us to make certain assumptions regarding environments. Note that this well-formedness results from the semantic rules for Java respectively Bahasa. However, we still need some more functionality. From now on we will handle the environment Γ as an implicit parameter wherever needed. This is justified by the fact that the methods defined work on a fixed program *p* that defines a fixed type environment Γ.

Additionally, we need to model the variable and method lookup mechanism of Java. The mechanism is implemented by means of four functions that accept a class or interface name and either a variable or a method name. *FieldDecl* returns for a class C and a field name f a pair indicating the class $\mathtt{C}'$ where f is declared and the type of the field. *FieldDecls* returns the set of such pairs for all fields named f that are declared in either C or one of its super classes. This is needed as *all* fields declared in one of the super classes of C classes are visible in C. Field access in Java (and therefore in Bahasa, too) is based on the static type of a variable only.

The function *MethodDecls* returns for a class C and a method name m a set of pairs $(\mathtt{C}', MT)$. Each of this pairs represents an implementation of m in $\mathtt{C}'$ with method type *MT*. These methods are the implementations that are visible in class C. Finally, *MethodSigs* returns all different method types of methods contained in *MethodDecls*. The last two functions also work on interface types.

Definition 3.5 *(Functions FieldDecl, FieldDecls, MethodDecls, MethodSigs)*
Let p be a Bahasa program and Γ the environment derived from p with a declaration for class C and interface I, i. e.

$$\begin{aligned}\Gamma(\mathtt{C}) &= (\mathtt{C}', \{\mathtt{I}_1, \cdots, \mathtt{I}_i\}, [\mathtt{v}_j \rightarrow (b_j, \mathtt{T}_j)]_{j=1\cdots n}, [(\mathtt{m}_k, MT_k) \rightarrow (b_k, s_k, \mathtt{f}_k)]_{k=1\cdots p}) \\ \Gamma(\mathtt{I}) &= (\epsilon, \{\mathtt{J}_1, \cdots, \mathtt{J}_j\}, [], [(\mathtt{im}_k, iMT_k) \rightarrow (ib_k, \mathtt{false}, [])]_{k=1\cdots p})\end{aligned}$$

We define the functions

$$\begin{aligned}\textit{FieldDecl}: &\ \textbf{\textit{Classes}} \times \textbf{\textit{VNames}} \rightarrow \textbf{\textit{Classes}} \times \textbf{\textit{Classes}} \\ \textit{FieldDecls}: &\ \textbf{\textit{Classes}} \times \textbf{\textit{VNames}} \rightarrow \mathcal{P}(\textbf{\textit{Classes}} \times \textbf{\textit{Classes}}) \\ \textit{MethodDecls}: &\ \textbf{\textit{OOTypes}} \times \textbf{\textit{MNames}} \rightarrow \mathcal{P}(\textbf{\textit{OOTypes}} \times \textbf{\textit{MTypes}}) \\ \textit{MethodSigs}: &\ \textbf{\textit{OOTypes}} \times \textbf{\textit{MNames}} \rightarrow \mathcal{P}(\textbf{\textit{MTypes}})\end{aligned}$$

by

$$\textit{FieldDecl}(\mathtt{C},\mathtt{v}) = \begin{cases} (\mathtt{C},\mathtt{T}_j) & \textit{if } \exists 1 \le j \le f : \mathtt{v} = \mathtt{v}_j \\ \textit{FieldDecl}(\mathtt{C}',\mathtt{v}) & \textit{if } \Gamma(\mathtt{C}) = (\mathtt{C}',\mathtt{I},f_f,f_m) \end{cases}$$

$$\textit{FieldDecls}(\mathtt{C},\mathtt{v}) = \begin{cases} \{\textit{FieldDecl}(\mathtt{Object},\mathtt{v})\} & \textit{if } \mathtt{C} = \mathtt{Object} \\ \{\textit{FieldDecl}(\mathtt{C},\mathtt{v})\} \cup \textit{FieldDecls}(\mathtt{C}',\mathtt{v}) & \textit{otherwise} \end{cases}$$

$$\begin{aligned} \textit{MethodDecls}(\mathtt{C},\mathtt{m}) &= \{(\mathtt{C},MT_j) | \textit{if } \exists 1 \le j \le p : \mathtt{m}_j = \mathtt{m}\} \cup \\ &\quad \{(\mathtt{C}'',MT'') | (\mathtt{C}'',MT'') \in \textit{MethodDecls}(\mathtt{C}',\mathtt{m}) \wedge \\ &\qquad \forall 1 \le j \le p : \mathtt{m} = \mathtt{m}_j \Rightarrow ArgTypes(MT_j) \neq \textit{ArgTypes}(MT'')\} \\ \textit{MethodDecls}(\mathtt{I},\mathtt{m}) &= \{(\mathtt{I},MT_j) | \textit{if } \exists 1 \le j \le p : \mathtt{m}_j = \mathtt{m}\} \cup \\ &\quad \{(\mathtt{I}',MT') | \exists 1 \le j \le n : (\mathtt{I}',MT') \in \textit{MethodDecls}(\mathtt{I}_j,\mathtt{m}) \wedge \\ &\qquad \forall 1 \le i \le p : m = m_i \Rightarrow \textit{ArgTypes}(MT_i) \neq \textit{ArgTypes}(MT')\} \\ \textit{MethodSigs}(\mathtt{C},\mathtt{m}) &= \{MT | \exists \mathtt{C}'' \in \textbf{\textit{Classes}} : (\mathtt{C}'',MT) \in \textit{MethodDecls}(\mathtt{C},\mathtt{m})\} \\ \textit{MethodSigs}(\mathtt{I},\mathtt{m}) &= \{MT | \exists \mathtt{I}' \in \textbf{\textit{Interfaces}} : (\mathtt{I}',MT) \in \textit{MethodDecls}(\mathtt{I},\mathtt{m})\} \end{aligned}$$

Note that all recursive definitions above are well-founded since a superclass $\mathtt{C}'$ *is only accessed if* $\mathtt{C} \neq \mathtt{Object}$*. This assures that the recursion will stop.*

Using these functions we are now able to dynamically identify in the semantics the method to be called at a given call site. This identification is based on the type of the object that the method is called on, the method name and the static types of the arguments.

Definition 3.6 *(more special, Functions Applicable, MostSpecial)*
Let p be a `Bahasa` *program,* Γ *the environment derived from p,* $\mathtt{m} \in$ ***MNames***, $\mathtt{C} \in OOTypes$, $\mathtt{T}_1,\cdots,\mathtt{T}_{n+1},\mathtt{T}'_1,\cdots,\mathtt{T}'_{n+1} \in$ ***Types*** *with* $AT = \mathtt{T}_1 \times \cdots \times \mathtt{T}_n$ *and* $AT' = \mathtt{T}'_1 \times \cdots \times \mathtt{T}'_n$.
A signature $(\mathtt{C},\mathtt{T}_1 \times \cdots \times \mathtt{T}_n \to \mathtt{T}_{n+1})$ *is defined to be* **more special** *than another signature* $(\mathtt{C}',\mathtt{T}'_1 \times \mathtt{T}'_n \to \mathtt{T}'_{n+1})$ *if* $\mathtt{C} \le_w \mathtt{C}' \wedge \forall 1 \le i \le n : \mathtt{T}_i \le_w \mathtt{T}'_i$*, written as*

$$(\mathtt{C},\mathtt{T}_1 \times \mathtt{T}_n \to \mathtt{T}_{n+1}) \preceq (\mathtt{C'},\mathtt{T}'_1 \times \mathtt{T}'_n \to \mathtt{T}'_{n+1})$$

The set of most special declarations for a call to method m with argument type AT based on an object with type C is defined by means of two functions $\textit{MostSpecial}, \textit{Applicable} :$ ***MNames*** $\times$ ***Classes*** $\times \bigcup_{n\ge 0}$ ***Types***$^n \to \mathcal{P}($***Classes*** $\times \bigcup_{n\ge 0}$ ***Types***$^n \times$ ***Term***$)$*:*

$$
\begin{aligned}
\mathit{Applicable}(\mathtt{m}, \mathtt{C}, \mathit{AT}) \;=\; \{\; & (\mathtt{C}', \mathit{MT}', \mathit{mbody}) \mid \\
& (\mathtt{C}', \mathit{MT}') \in \mathit{MethodDecls}(\mathtt{C}, \mathtt{m}) \\
& \textit{where } \mathit{MT}' = \mathit{AT}' \wedge \mathtt{T}_i \leq_w \mathtt{T}'_i, 1 \leq i \leq n \textit{ and} \\
& \mathtt{mbody} \textit{ is the body of } \mathtt{m}\} \\
\mathit{MostSpecial}(\mathtt{m}, \mathtt{C}, \mathit{AT}) \;=\; \{\; & (\mathtt{C}', \mathit{MT}', \mathit{mbody}) \mid \\
& (\mathtt{C}', \mathit{MT}', \mathit{mbody}) \in \mathit{Applicable}(\mathtt{m}, \mathtt{C}, \mathit{AT}) \wedge \\
& \big((\mathtt{C}'', \mathit{MT}'', \mathtt{mbody}_2) \in \mathit{Applicable}(\mathtt{m}, \mathtt{C}, \mathit{AT}) \wedge \\
& \qquad (\mathtt{C}'', \mathit{MT}'') \preceq (\mathtt{C}', \mathit{MT}') \\
& \quad \Rightarrow (\mathtt{C}'', \mathit{MT}'') = (\mathtt{C}', \mathit{MT})\big)\}
\end{aligned}
$$

The function *Applicable* is applied to a method `m`, a class `C`, and an argument type vector *AT*. It looks up all methods `m` that are defined in `C` or its super classes with an argument type AT' that *AT* can be widened to.

Applicable is used by method *MostSpecial*. Both functions are applied to the same arguments. *MostSpecial* checks the returned set for the method with the most special signature.

It is noteworthy that a `Bahasa` program *p* is only typeable if for every call statement the set $\mathit{MostSpecial}(\mathtt{m}, \mathtt{C}, \mathit{AT})$ has exactly one element, namely the method that will be called at run-time. Otherwise the call would be *ambiguous* and would result in a compile-time error. `C` is the run-time type of the object on which a method is called. For a more detailed comment on this see e. g. [DE1999].

Now we can finally specify what it means for an environment to be *well-formed*. A well-formed environment allows us to make certain assumptions regarding the program that induced it. The most important points are that the program defines all types referred to and that the methods defined by classes and interfaces adhere to the inheritance rules imposed by the language definition [GJS1996]. E. g., a malformed environment could contain variable declarations with undefined classes as variable types resulting in run-time errors when checking sub-type relationships at call sites.

The rules for this assertion $\Diamond_\Gamma$ are given in the following definition and will be explained subsequently. Note that $\Diamond_\Gamma$ is defined recursively, that is the parts of an environment are well-formed if the total environment is well-formed, and vice versa. This expresses that, e. g., the class hierarchy of the whole program is acceptable only if each class has an existing class as superclass.

Definition 3.7 *(Well-Formed Type-Environment)*
Let p be a program and Γ *the type-environment induced by p. We write* MT_j *for a method type* $(\mathtt{PT}_1 \times \cdots \times \mathtt{PT}_{a_i}) \rightarrow \mathtt{T}_j$.
Γ *is defined to be* **well-formed** *iff*

$$\forall \mathtt{C} \in \textbf{\textit{Classes}} : \Gamma(\mathtt{C})\Diamond_\Gamma \;\wedge\; \forall \mathtt{I} \in \textbf{\textit{Interfaces}} : \Gamma(\mathtt{I})\Diamond_\Gamma$$

where $assWF$ *means* well-formed with respect to Γ. $\Gamma(\texttt{C})\Diamond_\Gamma$ *and* $\Gamma(\texttt{I})\Diamond_\Gamma$ *are defined below. If* p *defines a class* $\texttt{C} \equiv$

```
class C extends C' implements I_1, ..., I_n {
        fmod_1 FT_1 f_1; ... fmod_{m_1} FT_m f_m;
        mmod_1 RT_1 m_1 (PT_(1,1) p_(1,1) ··· PT_(1,a_1) p_(1,a_1)) {
          VT_(1,1) lv_(1,1); ... VT_(1,v_1) lv_(1,v_1);
          Stmts }
        ...
        mmod_l RT_l m_l (PT_(l,1) p_(l,1) ··· PT_(l,a_l) p_(l,a_l)) {
          VT_(l,1) lv_(l,1); ... VT_(l,v_l) lv_(l,v_l);
          Stmts }}
```

then $\Gamma(\texttt{C})\Diamond_\Gamma$ *holds iff*

$m, n, l \geq 0$
$\texttt{C}' \not\sqsubseteq_\Gamma \texttt{C}$ $\quad \texttt{C}' \sqsubseteq_\Gamma \texttt{C}'$ $\quad \texttt{I}_j \leq_\Gamma \texttt{I}_j, 1 \leq j \leq n$
$\texttt{FT}_j \Diamond_{\texttt{VarType}}, 1 \leq j \leq m$ $\quad MT_j \Diamond_{\texttt{MethType}}\ 1 \leq j \leq l$
$\texttt{VT}_(i,j)\Diamond_{\texttt{VarType}}, 1 \leq i \leq l, 1 \leq j \leq v_i$
$\texttt{m}_i = \texttt{m}_j \Rightarrow\ i = j$ *or ArgTypes*$(MT_i) \neq$ *ArgTypes*$(MT_j), 1 \leq j, i \leq l$
$\forall 1 \leq j \leq l$: *MT* $\in$ *MethodSigs*$(\texttt{C}', \texttt{m}_j) \wedge$ *ArgTypes*$(MT) =$ *ArgTypes*(MT_j)
$\quad \Rightarrow$ *ResType*$(MT_j) =$ *ResType*(MT)
$\forall \texttt{m} \in$ ***MNames***$, 1 \leq j \leq n$: $AT \rightarrow T \in$ *MethodSigs*$(\texttt{I}_j, m)$
$\quad \Rightarrow \exists T'$: $AT \rightarrow T' \in$ *MethodSigs*$(\texttt{C}, m) \wedge T' \leq_w T$

If p *defines an interface* $\texttt{I} \equiv$

```
interface I extends I_1, ..., I_n {
        mmod_1 RT_1 m_1 (PT_(1,1) p_(1,1) ··· PT_(1,a_1) p_(1,a_1));
        ...
        mmod_l RT_l m_l (PT_(l,1) p_(l,1) ··· PT_(l,a_l) p_(l,a_l)); }
```

then $\Gamma(\texttt{I})\Diamond_\Gamma$ *holds iff*

$n, l \geq 0$
$\texttt{I} \not\leq_\Gamma \texttt{I}$ $\quad \texttt{I}_j \not\leq_\Gamma \texttt{I}$ $\quad \texttt{I}_j \leq_\Gamma \texttt{I}_j, 1 \leq j \leq n$
$MT_j \Diamond_{\texttt{MethType}}\ 1 \leq j \leq l$
$\texttt{m}_i = \texttt{m}_j \Rightarrow i = j$ *or ArgTypes*$(MT_i) \neq$ *ArgTypes*$(MT_j), 1 \leq j, i \leq l$
$\forall 1 \leq j \leq l, 1 \leq i \leq n$: *MT* $\in$ *MethodSigs*$(\Gamma, \texttt{I}_i, \texttt{m}_j) \wedge$ *ArgTypes*$(MT) =$ *ArgTypes*(MT_j)
$\quad \Rightarrow$ *ResType*$(MT_j) =$ *ResType*(MT)
$\forall \texttt{m} \in$ ***MNames***$, 1 \leq j, i \leq n$: $MT_1 \in$ *MethodSigs*$(\Gamma, \texttt{I}_i, \texttt{m})$,
$\quad MT_2 \in$ *MethodSigs*$(\texttt{I}_j, \texttt{m})$: *ArgTypes*$(MT_1) =$ *ArgTypes*(MT_2)
$\quad \Rightarrow$ *ResType*$(MT_1) =$ *ResType*(MT_2)

The two parts of this definition describe the requirements for a class respectively interface declaration to be well-formed. First we will look into the class part. Assume that p includes the class declaration given in the definition. The first requirements are that $\mathtt{C}'$ is not defined to be a subclass of $\mathtt{C}$ as this would introduce a circle in the class hierarchy. Moreover, the super class $\mathtt{C}'$ and all interfaces implemented by $\mathtt{C}$ must be defined. Additionally, all fields and variables declared in methods must have variable type and all methods must have a method type. The rest of the restrictions are also on methods and their types. A class may only define one method with name $\mathtt{m}$ and a specific argument type. If the super classes of $\mathtt{C}$ define a method with a name and an argument type that is the same as that of a method defined by $\mathtt{C}$ then the return types must be equal, too. Finally, if one of the interfaces that $\mathtt{C}$ implements contains a method declaration, then $\mathtt{C}$ must contain a method that has a result type $\mathtt{T}$ that can be widened to the interface method's return type.

The requirements for interfaces are almost identical, so we will skip their explanation. A detailed discussion of the requirements with respect to classes and interfaces can be found in [GJS1996].

In addition we define methods $Static_f$ and $Static_m$ that return the static flag that Γ associates with the field or method of the specified type. They use the methods just defined to identify the correct field or method.

Definition 3.8 *(Functions $Static_f$, $Static_m$)*
Assume $\Gamma(\mathtt{C}) = (\mathtt{C}_S, M_I, f_{fields}, f_{methods})$, $\mathtt{C} \in$ ***Classes***.
The functions $Static_f$: ***Classes*** $\times$ ***FNames*** $\rightarrow$ ***Bool*** *and* $Static_m$: ***Classes*** $\times$ ***MNames*** $\times$ ***MTypes*** $\rightarrow$ ***Bool*** *are defined as*

$$
\begin{array}{lcll}
Static_f(\mathtt{C},\mathtt{f}) & = & b_{static} & \textit{where} \quad (\mathtt{C}', \mathtt{T}_f) = \textit{FieldDecl}(\mathtt{C},\mathtt{f}) \wedge \\
 & & & \quad \Gamma(\mathtt{C}') = (\mathtt{C}'', M_I', f_{fields}', f_{methods}') \wedge \\
 & & & \quad f_{fields}'(\mathtt{f}) = (b_{static}, \mathtt{T}_f) \\
Static_m(\mathtt{C},\mathtt{m},MT) & = & b_{static} & \textit{where} \quad \{(\mathtt{C}', MT, \mathtt{mbody})\} = \\
 & & & \quad \textit{MostSpecial}(\mathtt{m}, \mathtt{C}', \textit{ArgTypes}(MT)) \wedge \\
 & & & \quad \Gamma(\mathtt{C}') = (\mathtt{C}'', M_I', f_{fields}', f_{methods}') \wedge \\
 & & & \quad f_{methods}(\mathtt{m}, MT) = (b_{static}, b_{constr}, f_{locals})
\end{array}
$$

3.4 The Run-Time Model

As mentioned above, Bahasa is divided into two parts. We have already introduced the type related half. Now the execution related part is going to be presented. The language $\mathtt{Bahasa}_r$ contains the already mentioned intermediate expressions and statements. Having constructed a well-defined environment, we use the functions Γ_V from Definition 3.3 and *MostSpecial* from Definition 3.6 to reconstruct Bahasa programs such that they comply to $\mathtt{Bahasa}_r$. The abstract syntax of $\mathtt{Bahasa}_r$ programs with auxiliary constructs is given in Table 3.6. The differences compared to Bahasa are

- Field access expressions are annotated with the static type of the variable used to access the field
- Method calls are annotated with the vector of static argument types
- Information about super classes, implemented interfaces, parameter and return types and variable declarations are deleted
- Additional statements `bind`, `end` and an additional expression for references to objects are added

On the one hand the static types are necessary to access the correct field (namely the one defined or inherited by the that class) or find the method with the matching signature respectively. On the other hand most of the type information is stored in Γ so it can safely be deleted.

The `bind` statement may only occur during evaluation of $\texttt{Bahasa}_r$ programs. It is used to bind the arguments passed to a method to the method's formal parameters. The body of the method called will be executed in this environment. After the method has finished the caller's environment is restored. Thus we can make the run-time stack of a `Bahasa` program explicit in syntax and semantics. Also references do only exist during run-time. They will be introduced briefly.

Before going on we define the domains for references, values and objects as well as the structures for representing variable environments and heaps.

Definition 3.9 ***(Addresses, References, Values, Objects, Env, Heap)***
Let p be a `Bahasa` program. We define the following domains and give typical meta variables we will use:

$$
\begin{array}{rll}
\iota_i \in & \textbf{\textit{Addresses}} & \textit{a countably infinite set of locations on the heap} \\
\iota_i^C \in & \textbf{\textit{References}} & := \textbf{\textit{Addresses}} \times \textbf{\textit{OOTypes}} \\
val \in & \textbf{\textit{Values}} & := \textbf{\textit{References}} \cup \{\texttt{true}, \texttt{false}, \mathit{nil}, \bullet\} \cup \mathbb{N} \\
obj \in & \textbf{\textit{Objects}} & := \textbf{\textit{Classes}} \times ((\textbf{\textit{FNames}} \times \textbf{\textit{Classes}}) \to \textbf{\textit{Values}}) \\
H \in & \textbf{\textit{Heap}} & := \textbf{\textit{References}} \to \textbf{\textit{Objects}} \\
\sigma \in & \textbf{\textit{Env}} & := (\textbf{\textit{VNames}} \cup \textbf{\textit{Classes}}) \to (\mathbb{N} \times \textbf{\textit{Values}})
\end{array}
$$

A reference will be written as ι_i^C where ι_i is the address and `C` is the run-time type of the object that is referenced. Objects are represented by pairs $(\texttt{C}, f_{\texttt{C}})$, where `C` is the type of the object and $f_{\texttt{C}}$ is a function mapping a field name and a class to a value. $f_{\texttt{C}}$ is defined for fields visible in `C` and the class $\texttt{C}'$ from which `C` inherits `f`. It returns the value stored in that field. Note further that a state maps a variable x to a pair (l, v) where v is the value actually stored in x and l is the label of the left hand side of the statement where v was assigned to x. This does not at all change the semantics of the language, but will allow us to simplify the proof of semantic correctness for our analysis in Chapter 4. σ is also defined on classes. It returns a pair consisting of label 0 and a reference to an object. This object is used to store the static fields of the class. The value $\bullet$ will

Program	:=	(*ClassDef*)*
ClassDef	:=	`class ClassId {` *MethodDefinition** `}`
MethodDefinition	:=	`MethodId (` (`ParamId`^{l_i})* `) {` *Stmts*^{l_r} `;` `}`
Block	:=	`{` *Stmts* `}` | *Stmt*;
Stmts	:=	*Stmts Stmt*; | ϵ
Stmt	:=	`if` *Expr* `then` *Block* `else` *Block* | `skip` | *Call*
	|	*Var*^{l_1} `=` *Rhs*^{l_2} | `return` [`VarName`^{l}]
	|	`bind` σ, H `in` *Stmts*`; end;`
Rhs	:=	*Var* | `new ClassName()` | *Call* | *Expr*
Call	:=	*Variable*^{l_o}`.MethodName(` *Variable*$^{l_i *}$ `)`
Var	:=	*Variable* | *Member*
Variable	:=	`VarName` | ι_i^C | `null` i an integer
Member	:=	`ClassName`^{l_o}`.[ClassName]MemberName`
	|	*Variable*^{l_o}`.[ClassName]MemberName`
Expr	:=	`true` | `false` | `c`

Table 3.6: `Bahasa`$_r$ syntax

Configuration	::=	$\langle t, \sigma, H \rangle$, $t \in$ **Term** $\cup \{\epsilon\}$, $\sigma \in$ **Env**, $H \in$ **Heap**
$\rightsquigarrow$	:	**Term** $\rightarrow Configuration \rightarrow Configuration$
$\rightsquigarrow_p$	:	$Configuration \rightarrow Configuration$

Table 3.7: Configurations for `Bahasa`$_r$

be used to unify the treatment of return statements; • will be returned by methods that originally do not return a value.

Table 3.7 describes the configurations used by the operational semantics. The environment $\sigma \in$ **Env** that we will use is flat, namely identifiers are mapped to primitive values or to references. Those references point to objects that are allocated on the heap $H \in$ **Heap**. An object and the reference to it are annotated by the object's dynamic class. The object itself consists of a sequence of labels and values. The labels identify the name of the field and the class where it has statically been defined. The operational semantics $\rightsquigarrow$ is a mapping from `Bahasa`$_r$ programs and configurations to configurations. As the program is fixed during execution, we abbreviate this as $\rightsquigarrow_P$, which maps configurations to configurations.

Before handling execution of (parts of) programs we first need to specify how execution effects states and objects on the heap. For this, we will define two functions *getfield* and *setfield*. It is

noteworthy how they model the way fields in objects are accessed in Java. For the identification of the field to be written or read only the static type of the variable whose field is accessed matters, not the dynamic type of the object stored in the variable. This is in contrast to method lookup. This is why the static type of the variable is passed as an argument to these functions and has been added to field accesses in $\texttt{Bahasa}_r$.

Definition 3.10 *(State and Heap Update, Functions getfield and setfield)*
Let $\texttt{obj}, \texttt{obj}' \in$ ***Objects***, $\texttt{obj} :=\ll (\texttt{l}_1, \texttt{C}_1) : \texttt{val}_1, \cdots, (\texttt{l}_n, \texttt{C}_n) : \texttt{val}_n \gg^{\texttt{C}'}$ *be an object of type* $\texttt{C}'$, $\sigma, \sigma' \in$ ***Env***, $\texttt{val} \in$ ***Values***, $l \in \mathbb{N}$, $\iota_i^{C''}, \iota_j^{C'''} \in$ ***References***, $\texttt{C} \in$ ***Classes***, $y, z \in$ ***VNames*** *identifiers and* $\texttt{f}, \texttt{l}_i \in$ ***FNames*** *field identifiers.*

We define the reading respectively writing access to field $\texttt{f}$ *of object* $\texttt{obj}$ *in class* $\texttt{C}$ *as*

$$\textit{getfield} : \textbf{\textit{Objects}} \times \textbf{\textit{FNames}} \times \textbf{\textit{Classes}} \to \textbf{\textit{Values}}$$

$$\textit{getfield}(\texttt{obj}, \texttt{f}, \texttt{C}) := \texttt{val}_i \quad \textit{if } \texttt{f} = \texttt{l}_i \ \wedge \ \textit{FieldDecl}(\texttt{C}, f) = (C_i, T)$$

$$\textit{setfield} : \textbf{\textit{Objects}} \times \textbf{\textit{FNames}} \times \textbf{\textit{Classes}} \times \textbf{\textit{Values}} \to \textbf{\textit{Objects}}$$

$$\textit{setfield}(\texttt{obj}, \texttt{f}, \texttt{C}, \texttt{val}) :=\ll (\texttt{l}_1, \texttt{C}_1) : \texttt{val}_1, \cdots, (\texttt{l}_i, \texttt{C}_i) : \texttt{val}, \cdots, (\texttt{l}_n, \texttt{C}_n) : \texttt{val}_n \gg^{C'} \quad \textit{if } \texttt{f} = \texttt{l}_i \ \wedge \ \textit{FieldDecl}(\texttt{C}, f) = (C_i, T)$$

The **update of values** *in the state* σ *and the heap* H *is defined by*

$$\sigma' := \sigma[\texttt{z} \mapsto (l, \texttt{val})] \Leftrightarrow \sigma'(\texttt{y}) = \begin{cases} (l, \texttt{val}) & \textit{if } \texttt{y} = \texttt{z} \\ \sigma(\texttt{y}) & \textit{otherwise} \end{cases}$$

$$H' := H[\iota_i^{C''} \mapsto \texttt{obj}'] \Leftrightarrow H'(\iota_j^{C'''}) = \begin{cases} \texttt{obj}' & \textit{if } \iota_j^{C'''} = \iota_i^{C''} \\ H(\iota_j^{C'''}) & \textit{otherwise} \end{cases}$$

Furthermore we will need to distinguish between those terms that cannot be further rewritten and those, that may not be further rewritten as they are on left hand sides of assignments.

Definition 3.11 *(Ground and L-Ground Expressions)*
A $\texttt{Bahasa}_r$*-term* t *is*

- **ground** *iff* t *is a primitive value or* $\bullet$ *or* $\exists i \in \mathbb{N}, \texttt{C} \in$ ***Classes*** $: t = \iota_i^C$
- **l-ground** *iff* $t = \texttt{id}$ *for some identifier* $\texttt{id} \in$ ***VNames***, *or there exist classes* $\texttt{C}, \texttt{C}' \in$ ***Classes***, *a field* $\texttt{f} \in$ ***FNames*** *and an integer* $i \in \mathbb{N}$ *such that* $t = \iota_i^{C'}.[\texttt{C}]\texttt{f}$.

[*Constant*] $$\frac{}{\langle \mathtt{c}, \sigma, H\rangle \rightsquigarrow_P \langle [\![\mathtt{c}]\!], \sigma, H\rangle}$$

[*Variable*] $$\frac{}{\langle \mathtt{VarName}, \sigma, H\rangle \rightsquigarrow_P \langle val, \sigma, H\rangle} \quad \sigma(\mathtt{VarName}) = (l, val)$$

[*Class*] $$\frac{}{\langle \mathtt{ClassName}, \sigma, H\rangle \rightsquigarrow_P \langle val, \sigma, H\rangle} \quad \sigma(\mathtt{ClassName}) = (l, val)$$

[*Field Access 1*] $$\frac{\langle Var, \sigma, H\rangle \rightsquigarrow_P \langle Var', \sigma, H\rangle}{\langle Var.[\mathtt{C}]\mathtt{f}, \sigma, H\rangle \rightsquigarrow_P \langle Var'.[\mathtt{C}]\mathtt{f}, \sigma, H\rangle}$$

[*Field Access 2*] $$\frac{}{\langle \iota_i^{C'}.[\mathtt{C}]\mathtt{f}, \sigma, H\rangle \rightsquigarrow_P \langle getfield(H(\iota_i^{C'}), \mathtt{f}, \mathtt{C}), \sigma, H\rangle}$$

[*Object Creation*] $$\frac{}{\langle \mathtt{new\ C()}, \sigma, H\rangle \rightsquigarrow_P \langle \iota_i^C, \sigma, H'\rangle} \quad \begin{array}{l}\iota_i^C \notin dom(H)\\ H' = H[\iota_i^C \mapsto \ll\gg^C]\end{array}$$

Table 3.8: Executing $\mathtt{Bahasa}_r$ expressions

3.5 Executing $\mathtt{Bahasa}_r$

We are now ready to present the rewriting rules for $\mathtt{Bahasa}_r$. The semantics chosen is a small step one. This will allow for a detailed presentation of the correctness proof in the next chapter.

First, we will introduce the rules for executing expressions, given in Table 3.8. These include evaluation of constants, access to variables and fields, rewriting of variables and expressions, and creation of classes and objects. Constants are simply evaluated by function $[\![\cdot]\!] : \textbf{Term} \rightarrow \textbf{Values}$. We do not touch basic types in any kind so we will not further specify expressions and their evaluation. The rule for variable access looks up the value of the variable in the environment. Note that the value is a pair consisting of the actual value `val` and the label l of the last definition of the variable. Of course only `val` is returned. For field accesses we first rewrite the variable to a reference and then pass the object stored in the heap to *getfield*. Finally, when creating an object of class `C` we pick a fresh reference, create an empty object of type `C` and pass the reference on. Note that we do not perform any garbage collection but assume an infinite heap where we always can obtain an unused reference.

Table 3.9 shows how to rewrite statements. Sequences of statements are evaluated from left to

[*sequence 1*] $$\frac{\langle \mathit{Stmt}, \sigma, H\rangle \rightsquigarrow_P \langle \epsilon, \sigma', H'\rangle}{\langle \mathit{Stmt;Stmts}, \sigma, H\rangle \rightsquigarrow_P \langle \mathit{Stmts}, \sigma', H'\rangle}$$

[*sequence 2*] $$\frac{\langle \mathit{Stmt}, \sigma, H\rangle \rightsquigarrow_P \langle \mathit{Stmt'}, \sigma', H'\rangle}{\langle \mathit{Stmt;Stmts}, \sigma, H\rangle \rightsquigarrow_P \langle \mathit{Stmt';Stmts}, \sigma', H'\rangle}$$

[*if then else 1*] $$\frac{\langle c, \sigma, H\rangle \rightsquigarrow_P \langle e', \sigma, H\rangle}{\begin{array}{l}\langle \texttt{if } c \texttt{ then } \mathit{Stmts}_1 \texttt{ else } \mathit{Stmts}_2, \sigma, H\rangle \\ \quad \rightsquigarrow_P \langle \texttt{if } e' \texttt{ then } \mathit{Stmts}_1 \texttt{ else } \mathit{Stmts}_2, \sigma, H\rangle\end{array}}$$

[*if then else 2*] $$\frac{}{\begin{array}{l}\langle \texttt{if true then } \mathit{Stmts}_1 \texttt{ else } \mathit{Stmts}_2\ , \sigma, H\rangle \\ \quad \rightsquigarrow_P \langle \mathit{Stmts}_1, \sigma, H\rangle\end{array}}$$

[*if then else 3*] $$\frac{}{\begin{array}{l}\langle \texttt{if false then } \mathit{Stmts}_1 \texttt{ else } \mathit{Stmts}_2\ , \sigma, H\rangle \\ \quad \rightsquigarrow_P \langle \mathit{Stmts}_2, \sigma, H\rangle\end{array}}$$

[*bind 1*] $$\frac{\langle \mathit{Stmts}, \sigma', H'\rangle \rightsquigarrow_P \langle \mathit{Stmts'}, \sigma'', H''\rangle}{\begin{array}{l}\langle \texttt{bind } \sigma', H' \texttt{ in } \mathit{Stmts}, \sigma, H\rangle \\ \quad \rightsquigarrow_P \langle \texttt{bind } \sigma'', H'' \texttt{ in } \mathit{Stmts'}, \sigma, H\rangle\end{array}}$$

[*bind 2*] $$\frac{}{\langle \texttt{bind } \sigma', H' \texttt{ in end}, \sigma, H\rangle \rightsquigarrow_P \langle \epsilon, \sigma, H'\rangle}$$

[*bind 3*] $$\frac{\langle \mathit{val}, \sigma, H\rangle \rightsquigarrow_P \langle \mathit{val'}, \sigma, H\rangle}{\begin{array}{l}\langle \texttt{bind } \sigma', H' \texttt{ in return } \mathit{val}\texttt{; } \mathit{Stmts}\texttt{; end}, \sigma, H\rangle \\ \quad \rightsquigarrow_P \langle \mathit{val'}, \sigma, H'\rangle\end{array}}$$ $\mathit{val'}$ is ground

[*return 1*] $$\frac{\langle \mathit{Var}, \sigma, H\rangle \rightsquigarrow_P \langle e, \sigma, H\rangle}{\langle \texttt{return } \mathit{Var}, \sigma, H\rangle \rightsquigarrow_P \langle \texttt{return } e, \sigma, H\rangle}$$

[*return 2*] $$\frac{}{\langle \texttt{return}, \sigma, H\rangle \rightsquigarrow_P \langle \texttt{return } \bullet, \sigma, H\rangle}$$

[*skip*] $$\frac{}{\langle \texttt{skip}, \sigma, H\rangle \rightsquigarrow_P \langle \epsilon, \sigma, H\rangle}$$

Table 3.9: Executing $\texttt{Bahasa}_r$ statements

[*assign 1*]	$\dfrac{\langle Var, \sigma, H\rangle \rightsquigarrow_P \langle Var', \sigma, H\rangle}{\langle Var = e, \sigma, \mathtt{H}\rangle \rightsquigarrow_{\mathtt{P}} \langle Var' = e, \sigma, \mathtt{H}\rangle}$	*Var* is not l-ground
[*assign 2*]	$\dfrac{\langle e, \sigma, H\rangle \rightsquigarrow_P \langle e', \sigma, H\rangle}{\langle Var = e, \sigma, \mathtt{H}\rangle \rightsquigarrow_{\mathtt{P}} \langle Var = e', \sigma, \mathtt{H}\rangle}$	*Var* is l-ground
[*assign 3*]	$\dfrac{}{\langle \mathtt{id}^{l_1} \ \mathtt{=}\ val, \sigma, H\rangle \rightsquigarrow_P \langle \epsilon, \sigma[\mathtt{id} \mapsto (l_1, val)], H\rangle}$	*val* is ground, $\mathtt{id} \in$ **VNames**
[*assign 4*]	$\dfrac{}{\langle \iota_i^{C'} \mathtt{.[C]f} \ \mathtt{=}\ val, \sigma, H\rangle \rightsquigarrow_P \langle \epsilon, \sigma, H'\rangle}$	*val* is ground $H(\iota_i^{C'}) = \ll \cdots \gg^{C'}$ $H' = H[\iota_i^{C'} \rightarrow setfield(H(\iota_i^{C'}), \mathtt{f}, \mathtt{C}, val)]$

Table 3.10: Rules for assignments in $\mathtt{Bahasa}_r$

[*call 1*]	$\dfrac{\langle e_k, \sigma, H\rangle \rightsquigarrow_P \langle e'_k, \sigma, H\rangle}{\langle val_1\mathtt{.[AT]m}(val_2, \cdots, val_{k-1}, e_k, \cdots, e_n), \sigma, H\rangle \rightsquigarrow_P \langle val_1\mathtt{.[AT]m}(val_2, \cdots, val_{k-1}, e'_k, \cdots, e_n), \sigma, H\rangle}$	val_j is ground, $1 \le j < k \le n$
[*call 2*]	$\dfrac{}{\langle \iota_i^C\mathtt{.[AT]m}(val_2, \cdots, val_n), \sigma, H\rangle \rightsquigarrow_P \langle (\mathtt{bind}\ \sigma', H\ \mathtt{in}\ Stmts\ \mathtt{end})^{l_r}, \sigma, H\rangle}$	$n \ge 2$, $val_2 = \iota_i^C$, val_i is ground for $2 \le i \le n$, $Static(\mathtt{C}', \mathtt{m}, MT') = \mathtt{false}$, $AT = \mathtt{T}_3 \times \cdots \times \mathtt{T}_n$, $MostSpecial(\mathtt{m}, \mathtt{C}, AT) = \{(\mathtt{C}', MT', \mathtt{T}_r\ \mathtt{m}(\mathtt{T}_2\ \mathtt{x}_2{}^{lp_2}, \cdots, \mathtt{T}_n\ \mathtt{x}_n{}^{lp_n})\ \{\ Stmts\ \}^{l_r})\}$, $H(\iota_i^C) = \ll \cdots \gg^C$, $\sigma' = \sigma[\mathtt{x}_2 \mapsto val_2] \cdots [\mathtt{x}_n \mapsto val_n]$
[*call 3*]	$\dfrac{}{\langle \iota_i^C\mathtt{.[AT]m}(\mathtt{val}_2, \cdots, \mathtt{val}_n), \sigma, H\rangle \rightsquigarrow_P \langle (\mathtt{bind}\ \sigma', H\ \mathtt{in}\ Stmts\ \mathtt{end})^{l_r}, \sigma, H\rangle}$	$n \ge 1$, val_i is ground for $2 \le i \le n$, $Static(\mathtt{C}', \mathtt{m}, MT') = \mathtt{true}$, $AT = \mathtt{T}_2 \times \cdots \times \mathtt{T}_n$ $MostSpecial(\mathtt{m}, \mathtt{C}, AT) = \{(\mathtt{C}', MT', \mathtt{T}_r\ \mathtt{m}(\mathtt{T}_2\ \mathtt{x}_2{}^{lp_2}, \cdots, \mathtt{T}_n\ \mathtt{x}_n{}^{lp_n})\ \{Stmts\}^{l_r})\}$, $H(\iota_i^C) = \ll \cdots \gg^C$, $\sigma' = \sigma[\mathtt{x}_2 \mapsto val_2] \cdots [\mathtt{x}_n \mapsto val_n]$

Table 3.11: Rules for method calls in $\mathtt{Bahasa}_r$

right. In `if then else` statements the expression is evaluated first and depending on the value the `then` or the `else` branch is chosen. While we do not actually support expressions the rule has been added for reasons of completeness. Remember that *c* represents all expressions with simple type. `if ··· then` statements can be modeled by choosing the `skip` statement in one branch. The `bind` statement will evaluate its statements in the bound environment and heap. As already said this will be used to deal with method invocation and will encode the run-time stack into the program. The `return` statement is handled by `bind`, too. The value to be returned and the heap are passed back to the caller and the local environment of the returning method is disposed. Of course, garbage will occur whenever the only references to an object have been stored in local variables. Because of this special handling the return statements just rewrite the returned value until it is ground. If the method does have return type `void` then the special value $\bullet$ is inserted. The `skip` statement is consumed without any effect.

The rules in Table 3.10 describe how to execute assignments. First the left hand side is evaluated to a *l-ground* value, that is either a local variable or a reference. The next step rewrites the right hand side to a *ground* term. This term is then assigned to the left hand side, updating the state σ or the heap H accordingly. Note how the label of the right hand side is stored in σ for assignments to local variables.

Note that there are no rules to directly assign values to or dereference references. This is according to the design and security principles of Java [GJS1996]. References are only visible and accessible from inside the language execution process. This has been the motivation for the semantics design we have chosen, namely to add references only as intermediate expressions to $\texttt{Bahasa}_r$.

One important feature of programming languages is still missing – the evaluation of method calls. The rewriting rules (c. f. Table 3.11) first evaluate receiver and argument expressions from left to right. The second and third rule describe how to perform the actual call for either instance or class methods. The difference is that, no `this` reference is passed to static, i. e. class methods. Those methods are not called on an object but directly. Both rules result in the `bind` statement added to the configuration. Note how we save the label of the method body by assigning it to the `bind` construct.

For instance methods the method lookup is based on the *dynamic* type of the receiving object and the *static* types of the arguments, i. e. the *static* method type. The value val_1 represents the object. As that value is passed as the `this` pointer to the method, val_2 must have the same value and n must be at least 2. The result of *MostSpecial* is the one method to be called. This method provides us with the names of parameters and the statements to perform. We then compute the new state by binding the parameters to the arguments.

For static methods the lookup is just based on the *static* method type. There are no further restrictions, that is we look up the method to be called in the result of the call to *MostSpecial*. Then the new state is computed and `bind` is added to the configuration.

There still remains the question how to start the evaluation of a program. In Java the entry-point of a program is a method with declaration `public static void main(String args[])`

```
class Election extends Object{
    int count;
    void Election(Election ta[1]) {
        ta[2].[]Object(ta[3]);
        ta[4].[Election]count = 1;
        return;
    }
    void tick(Election tb[5]) {
        int x;
        x = tb[6].[Election]count;
        x = x+1;
        tb[7].[Election]count = x;
        return;
    }
}
class Election2 extend Election{
    void Election2(Election2 tc[8]) {
        tc[9].[]Election(tc[10]);
        return;
    }
    void tick(Election2 td[11]) {
        int y;
        y = td[12].[Election2]count;
        y = y+2;
        td[13].[Election2]count = y;
        return;
    }
}
```

```
class Object {
    void Object(Object te[14]) { return; }
}
class App extends Object{
    static void main() {
        Election candA,candB;
        candA[15] = new Election()[16];
        candA[17].[]Election(candA[18]);
        candB[19] = new Election2()[20];
        candB[21].[]Election2(candB[22]);
        candB[23].[]tick(candB[24]);
        candA[25].[]tick(candA[26]);
        return;
    }
}
```

Figure 3.1: Example program

{ *Stmts* }. So evaluation is initialized with $\sigma = [args \mapsto \iota_i^{String}]$ and $H = [\iota_i^{String} \mapsto \ll \cdots \gg^{String}]$ and the body of `main` as statements.

In order to give an overview of how the semantics works we will now examine an example program.

Example 3.6 *(Execution of $Bahasa_r$)*
Assume the example program given in Figure 3.1 which shows the $\texttt{Bahasa}_r$ version of the slightly modified examples in Section 3.1. The program is supposed to implement a computer based election system in order to make the counting procedure reliable.

It implements classes `App`, `Election` and `Election2`. The class `Election` implements a method `tick` that shall be called whenever a ballot voting for a candidate is found. The class

`Election2` extends `Election`. Both classes are supposed to be an object creation factory for objects representing candidates. In order to give the programmer's favorite candidate a boost, class `Election2` overwrites the method `tick` in order to increase the count by 2.

The class `App` creates an object of every candidate class and evaluates the two ballots.

As can be seen, the member accesses and method calls have already been rewritten to contain the static types of the object and the static argument types. Note that the argument types do not contain the type of the *this* pointers but are empty, as non of the functions gets an additional argument. Those pointers additionally have been renamed to unique names as required for `Bahasa` programs. The environment Γ induced by the program is given below. Note that for method types with no arguments we use `void` for the empty type.

$$
\begin{array}{lll}
\Gamma = [\ \texttt{Object} & \rightarrow & (\epsilon, [], [], [(\texttt{Object}, \texttt{void} \rightarrow \texttt{cstr}) \rightarrow (\texttt{false}, [\texttt{te} \rightarrow \texttt{Object}])]), \\
\texttt{Election} & \rightarrow & (\texttt{Object}, [], [\texttt{count} \rightarrow (\texttt{false}, \texttt{int})], \\
 & & \ [(\texttt{Election}, \texttt{void} \rightarrow \texttt{cstr}) \rightarrow (\texttt{false}, \texttt{true}, [\texttt{ta} \rightarrow \texttt{Election}]), \\
 & & \ (\texttt{tick}, \texttt{void} \rightarrow \texttt{void}) \rightarrow (\texttt{false}, \texttt{false}, [\texttt{tb} \rightarrow \texttt{Election}])]), \\
\texttt{Election2} & \rightarrow & (\texttt{Election}, [], [], \\
 & & \ [(\texttt{Election2}, \texttt{void} \rightarrow \texttt{cstr}) \rightarrow (\texttt{false}, \texttt{true}, [\texttt{tc} \rightarrow \texttt{Election2}]), \\
 & & \ (\texttt{tick}, \texttt{void} \rightarrow \texttt{void}) \rightarrow (\texttt{false}, \texttt{false}, [\texttt{td} \rightarrow \texttt{Election2}])]), \\
\texttt{App} & \rightarrow & (\texttt{Object}, [], [], \\
 & & \ [(\texttt{main}, \texttt{void} \rightarrow \texttt{void}) \rightarrow (\texttt{true}, \texttt{false}, [\texttt{candA} \rightarrow \texttt{Election}, \\
 & & \qquad \texttt{candB} \rightarrow \texttt{Election}])])]
\end{array}
$$

We will now execute the program. The program entry point is the method *main* in class *App*, so initially a `bind` statement with this procedures body will be in the configuration. Where relevant, the heap, the environment and the statement to execute next are given. In order to avoid line breaks we abbreviate `Election` by `E` and `Election2` by `E2` in states and heaps.

$\langle \texttt{bind}\ \sigma_1, H_1\ \texttt{in}\ \texttt{candA}^{15}\ \texttt{=}\ \texttt{new}\ \texttt{Election()}^{16}\texttt{;}\cdots\texttt{;}\ \texttt{end}, \sigma_0, H_0 \rangle$

Initially the environment and the heap are empty, that is $\sigma_1 = []$

$H_1 = []$

Now a reference to an `Election` object is created and stored in `candA`. This results in

$\sigma_2 = [\texttt{candA} \mapsto (15, \iota_1^E)]$

$H_2 = [\iota_1^E \mapsto \ll \texttt{count}\ \texttt{Election} : 0 \gg^E]$

$\langle \texttt{bind}\ \sigma_2, H_2\ \texttt{in}\ \texttt{candA}^{17}\texttt{.[]Election(candA}^{18}\texttt{);}\cdots\texttt{;}\ \texttt{end}, \sigma_0, H_0 \rangle$

The two occurrences of `candA` are rewritten to ground terms and constructor `Election` is called, resulting in a new state σ_3 that binds the local variables and parameters.

$\langle \texttt{bind}\ \sigma_2, H_2\ \texttt{in}\ (\iota_1^E)^{17}\texttt{.[]Election((}\iota_1^E\texttt{)}^{18}\texttt{);}\cdots\texttt{;}\ \texttt{end}, \sigma_0, H_0 \rangle$

$\sigma_3 = [\texttt{ta} \mapsto (1, \iota_1^E)]$

$\langle$`bind` σ_2, H_2 `in`
`bind` σ_3, H_2 `in` `ta`^{2}`.[]Object(ta`^{3}`);`$\cdots$`; end`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_0\rangle$

Now the two occurrences of `ta` need to be rewritten to ground terms and the constructor `Object` is called, again introducing a new state σ_4.

$\langle$`bind` σ_2, H_2 `in`
`bind` σ_3, H_2 `in` $(\iota_1^E)^2$`.[]Object(`$(\iota_1^E)^3$`);`$\cdots$`; end`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_0\rangle$

$\sigma_4 = [\texttt{te} \mapsto (14, \iota_1^E)]$

$\langle$`bind` σ_2, H_2 `in`
`bind` σ_3, H_2 `in`
`bind` σ_4, H_2 `in return;end`
`ta`^{4}`.[Election]count = 1;return;end`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_0\rangle$

As `Object` returns, the local state σ_4 is disposed and the innermost bind disappears.

$\langle$`bind` σ_2, H_2 `in`
`bind` σ_3, H_2 `in`
`ta`^{4}`.[Election]count = 1;return;end`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_0\rangle$

$\langle$`bind` σ_2, H_2 `in`
`bind` σ_3, H_2 `in`
$(\iota_1^E)^4$`.[Election]count = 1;return;end`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_0\rangle$

Now in the object referenced by ι_1^E field `count` is updated to 1, resulting in a new heap H_3. After the assignment the constructor returns and the execution of `main` is continued.

$H_3 = [\iota_1^E \mapsto \ll \texttt{count Election} : 1 \gg^E]$

$\langle$`bind` σ_2, H_3 `in`
`candB`^{19} `= new Election2()`^{20}`;`$\cdots$`; end`$, \sigma_0, H_3\rangle$

The execution continues by creating a fresh object of type `Election2` and calling the constructor of that class. This is almost identical to the initialization just shown, so we only give the resulting state, heap and configuration.

$\sigma_5 = [\texttt{candA} \mapsto (15, \iota_1^E), \texttt{candB} \mapsto (19, \iota_2^{E2})]$
$H_5 = [\iota_1^E \mapsto\ll \texttt{count Election} : 1 \gg^E, \iota_2^{E2} \mapsto\ll \texttt{count Election} : 1 \gg^{E2}]$

$\langle \texttt{bind } \sigma_5, H_5 \texttt{ in}$
$\quad \texttt{candB}^{23}\texttt{.[]tick(candB}^{24}\texttt{);candA}^{25}\texttt{.[]tick(candA}^{26}\texttt{);return;end}, \sigma_0, H_5\rangle$

$\langle \texttt{bind } \sigma_5, H_5 \texttt{ in}$
$\quad (\iota_2^{E2})^{23}\texttt{.[]tick}((\iota_2^{E2})^{24}\texttt{);candA}^{25}\texttt{.[]tick(candA}^{26}\texttt{);return;end}, \sigma_0, H_5\rangle$

Using Γ, the argument type `[]` and the run-time type `Election2` of reference ι_2^{E2} the target method of this call can uniquely be identified by means of *MostSpecial*. It is the implementation of `tick()` in class `Election2`.

$\sigma_6 = [\texttt{td} \mapsto (11, \iota_2^{E2})]$

$\langle \texttt{bind } \sigma_2, H_5 \texttt{ in}$
$\quad \texttt{bind } \sigma_6, H_5 \texttt{ in}$
$\quad\quad \texttt{y = td}^{12}\texttt{.[Election2]count;} \cdots \texttt{; end}$
$\quad \texttt{candB}^{19} \texttt{ = new Election2()}^{20}\texttt{;} \cdots \texttt{; end}, \sigma_0, H_0\rangle$

As can be seen from the source code this will add 2 to the member `count` of the object referenced by ι_2^{E2}. Thus the result after this method will be a new heap.

$H_6 = [\iota_1^E \mapsto\ll \texttt{count Election} : 1 \gg^E, \iota_2^{E2} \mapsto\ll \texttt{count Election} : 3 \gg^{E2}]$

$\langle \texttt{bind } \sigma_5, H_6 \texttt{ in}$
$\quad \texttt{candA}^{25}\texttt{.[]tick(candA}^{26}\texttt{);return;end}, \sigma_0, H_5\rangle$

$\langle \texttt{bind } \sigma_5, H_6 \texttt{ in}$
$\quad (\iota_1^E)^{25}\texttt{.[]tick}((\iota_1^E)^{26}\texttt{);return;end}, \sigma_0, H_5\rangle$

Again the target method of this call can uniquely be identified. Based on the object referenced by ι_1^E it is the implementation of `tick` in class `Election`. The rewriting takes place just like before, so we just give the final configuration before returning from `main`.

$\langle \texttt{bind } \sigma_5, H_6 \texttt{ in}$
$\quad \bullet\texttt{end}, \sigma_0, H_5\rangle$

$\sigma_5 = [\texttt{candA} \mapsto (15, \iota_1^E), \texttt{candB} \mapsto (19, \iota_2^{E2})]$
$H_6 = [\iota_1^E \mapsto\ll \texttt{count Election} : 2 \gg^E, \iota_2^{E2} \mapsto\ll \texttt{count Election} : 3 \gg^{E2}]$

From σ_5 and H_6 we get that the candidate represented by the object stored in `candA` has got 2 votes while the one represented by the object stored in `candB` has got 3 votes.

Chapter 4

Generating Symbolic Constraints

Theory is the sleep of reason.

Peter Sloterdijk

In Section 2.1.2 we have given a very brief introduction to control flow analysis. We will now fill in the pieces left out. Our control flow analysis will compute for each *interesting* subexpression of a `Bahasa` program p a set of classes which represent the possible run-time types of objects associated with the expression. *Interesting* are those expressions that may result in a class or interface type. Following the syntax given in Table 3.1, in a `Bahasa` program p all such points already have been assigned a label. Using these we will start by taking the Structural Operational Semantics from Chapter 3. It can be used to analyze all programs, directed by program evaluation as performed guided by the rewriting rules. This approach will reanalyze method bodies at every call site. It will allow a proof of semantic correctness, but cannot immediately be implemented as an efficient algorithm. It is not even computable. This is why we introduce a syntax directed specification in Section 4.3. It improves on the above by only analyzing each method body once. Using this we will be able to develop a constraint-based solution that will be used as the basis of our algorithm.

It is noteworthy that the analysis we describe is a two step approach. Given the program as input, the control flow analysis will generate a set of constraints. These constraints describe the flow of data through the program. They are then solved in a second step to produce a solution.

We start with summarizing key properties of our approach.

4.1 Key Properties

In order to allow for a better understanding of some design decisions, we will first give a short overview of our approach. Not all points may be understood on the first reading – explanations

will be given in this or the next chapter.

As already stated, our approach is a control flow analysis. We label every program point that may take on a class value. In order to allow for computing a solution, we abstract the program by a set of inclusion constraints between these labels. For every method we can generate a set of such constraints, describing the data flow in that method. As long as no class members are accessed or methods are called, these sets contain only constraints that model data flow between labels of local variables or parameters. That is, no constraints exist that model flow between two different methods.

In order to keep this property, we take special care of handling the object-oriented features in our language, that is access to class or instance members and calling of methods. Chapter 3 explains how these features are based on the run-time type of objects. That is, depending on the run-time type of an object, different members may be accessed or different methods may be called. The standard approach as presented in [NNH1999] is to generate *conditional* constraints, e. g., $\mathtt{A} \in \widehat{\chi}(l_o) \Rightarrow \widehat{\chi}(l_1) \subseteq \widehat{\chi}(l_2)$. These constraints check whether a type is element of a set – if yes, the inclusion on the right hand side is evaluated. However, these constraints establish connections between methods. A typical application is to let the values of actual parameters flow to the formal parameters of a method. Since the constraints are generated statically before solving them, we would need to generate a whole set for every call site, for every parameter of every possibly called method. This would result in a huge number of constraints; for every call and every argument the solver would need to check the condition in order to decide whether to propagate data or not.

In order to avoid this, we introduce *functor constraints*. These represent the object-oriented features. That is, we have a `LOAD`, a `STORE` and a `CALL` functor. Each of them gets as arguments all information needed to simulate the run-time behavior of calls and member access. As a result, the constraints generated for a method remain unconnected. This can easily be seen since, e. g. , the `CALL` functor only depends on information from the method itself.

In a second phase, the constraint solver evaluates the constraints. Only at this stage the functor constraints are evaluated. As a result, new constraints may be added to the constraint system. Now, previously unconnected constraint sets can get connected. This may take place, e. g., if a `CALL` functor determines a method to be callable and adds constraints to connect actual to formal parameters.

This approach has two major advantages. It is *demand driven*, that is, the constraints for a procedure only need to be generated when the method actually is detected to be callable. Furthermore, it is *modular*. Since the constraints are not connected, we can precompute them for methods and can store them separately. When the constraints for a method shall be added, we only need to check whether the precomputed constraints are still valid. If yes, we can insert them. Only in the case that the method has changed we need to recompute its constraints.

As pointed out in Chapter 3, Java programs use large standard libraries. The methods in those libraries are rarely changed and can easily be preprocessed.

4.2 Abstract Control Flow Analysis

We will now describe how to generate constraints from a program. This is done by means of a 0-CFA. That is, we are flow- but not context-sensitive.

As seen in the previous chapter, program points that may have class type are labeled. By virtue of these labels we compute for each using occurrence of a variable the set of reaching definitions. This is a simple intra-procedural analysis. Its result is a function $rd \in RD : \mathbb{N} \to \mathcal{P}(\mathbb{N})$. Sometimes we also want to speak about definition-use chains in program executions. We compute for each defining occurrence x^l the set $du \in DU$: **VNames** $\times \mathbb{N} \to \mathcal{P}(\mathbb{N})$. The set du contains the labels of all occurences of x that may use the value assigned at label l.

We start by defining the abstract semantic domains.

Definition 4.1 *(Abstract Semantic Domains)*
Let p be a `Bahasa` program. We define a number of semantic domains:

$$\begin{array}{ll} \widehat{\textbf{\textit{Values}}} & := \mathcal{P}(\textbf{\textit{Classes}}) \\ \widehat{\textbf{\textit{Cache}}} & := \mathbb{N} \to \widehat{\textbf{\textit{Values}}} \\ \widehat{\textbf{\textit{Heap}}} & := \textbf{\textit{Classes}} \times \textbf{\textit{FNames}} \times \textbf{\textit{Classes}} \to \widehat{\textbf{\textit{Values}}} \end{array}$$

The abstract values $\widehat{\textbf{Values}}$ that we will trace are sets of classes. These sets are associated with program points. Each set may be euqal to or a superset of the classes of objects reaching that point. Abstract caches $\widehat{\chi} \in \widehat{\textbf{Cache}}$, which map labels to abstract values, will handle all labeled subexpressions of the program being analyzed. Finally, abstract heaps $\widehat{H} \in \widehat{\textbf{Heap}}$ are used to store the values computed for class and instance members. The arguments are the run-time type, the member name and the static type of the expression used to access a field.

4.2.1 Abstractions

Before developing the abstract control flow analysis we will first describe and motivate the abstractions used. The entities that we will associate abstract values with are *local variables* and *class* and *instance members*.

Local Variables and Parameters

Local Variables are modeled rather fine grained. As introduced in the previous chapter, all their occurrences are labeled. This allows to model the flow between uses and definitions of local variables and parameters quite accurately. The interprocedural abstraction is performed by unifying all call contexts of a procedure as described in Section 2.1.3.

Class and Instance Members

The situation with class and instance members is rather different. Those are the constructs that allow multiple threads to communicate with each other. That is, the finer we model members, the more dependencies between using and writing accesses we would have to add. However, this would increase the analysis' complexity. This is why we abstract all instances of a class by a single entity. Of course, this is a very coarse abstraction, adding a lot of imprecision. But on the other side by doing so we also gain efficiency, as we do not need to perform any kind of points-to analysis as described in [RMR2001]. The problem is that we would need to know, which variables point to which object in order to update the correct fields in our abstraction. This implies knowledge on the layout of the heap, which is a hard problem [SRW1999].

4.2.2 The Acceptability Relation $\models$

We are now ready to describe our CFA, which is a 0-CFA [NNH1999]. We first specify an acceptability relation

$$\models \subseteq (\widehat{\textbf{Cache}} \times \widehat{\textbf{Heap}} \times \textbf{Term})$$

that will help us to identify those solutions that specify *acceptable* values for the labeled points.

Tables 4.1 and 4.2 define the relation. The expression

$$(\widehat{\chi}, \widehat{H}) \models t$$

means that the values stored by $\widehat{\chi}$ and $\widehat{H}$ fulfill the requirements imposed by *t*, i. e. they model the possible flow of values into *t*.

The first rule handles expressions of all primitive types. Remember that we have replaced those by the constant `c`. This is plausible as we only care about class types. Expressions and values of primitive types shall always be accepted. The next two clauses model how the different possible forms of *Variable* shall be handled. $[\widehat{\textit{VarName}}]$ imposes that at each labeled use of a variable the labels from all defining occurrences of this variable must be collected. This can easily be established by accessing the information computed before; $rd(l)$ contains all labels that identify defining occurrences of a variable.

The rule $[\widehat{\textit{ClassName}}]$ is needed to allow a uniform handling of class members by the next rule.

The rule $[\widehat{\textit{Member}}]$ checks for correct access to member fields of classes and instances. Therefore, a solution must first assure that the flow to the label l_o of the variable is established. For each of the classes the values from the abstract heap $\widehat{H}$ must flow to the cache of label l. As the field to access is determined based on the static type of the variable, this type is passed as an argument to $\widehat{H}$.

The clause $[\widehat{\textit{Create}}]$ handles occurrences of object creation on the right hand side of assignments. It does so by checking that the singleton set with the name of the created class is added to the set of the right hand side, labeled l.

$[\widehat{Constant}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{c}$
			always
$[\widehat{VarName}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{VarName}^l$
		iff	$\forall l' \in rd(l) : \widehat{\chi}(l') \subseteq \widehat{\chi}(l)$
$[\widehat{ClassName}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{ClassName}^l$
		iff	$\{\texttt{ClassName}\} \subseteq \widehat{\chi}(l)$
$[\widehat{Member}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$(\texttt{Var}^{l_o}\,.\,\texttt{[C]MemberName})^l$
		iff	$(\widehat{\chi},\widehat{H}) \models \texttt{Var}^{l_o} \ \wedge\ \forall \texttt{C}' \in \widehat{\chi}(l_o) :$
			$\widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \subseteq \widehat{\chi}(l)$
$[\widehat{Create}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$(\texttt{new ClassName}())^l$
		iff	$\{\texttt{ClassName}\} \subseteq \widehat{\chi}(l)$
$[\widehat{Sequence}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$Stmt\,;\ Stmts$
		iff	$(\widehat{\chi},\widehat{H}) \models Stmt \ \wedge\ (\widehat{\chi},\widehat{H}) \models Stmts$
$\widehat{[If]}$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{if}\ Expr\ \texttt{then}\ Block_1\ \texttt{else}\ Block_2$
		iff	$(\widehat{\chi},\widehat{H}) \models Block_1 \ \wedge\ (\widehat{\chi},\widehat{H}) \models Block_2$
$[\widehat{Return\ 1}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{return Variable}^l$
		iff	$(\widehat{\chi},\widehat{H}) \models \texttt{Variable}^l \ \wedge\ \widehat{\chi}(l) \subseteq \widehat{\chi}(l_r)$
			where l_r is the label of the method body analyzed
$[\widehat{Return\ 2}]$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{return}$
			always
$\widehat{[Skip]}$	$(\widehat{\chi},\widehat{H})$	$\models$	$\texttt{skip}$
			always

Table 4.1: Acceptability relation $\models$

$$
\begin{array}{lll}
\widehat{[\textit{Assign 1}]} & (\widehat{\chi},\widehat{H}) \models \texttt{VarName}^{l_1} \texttt{ = } rhs^{l_2} \\
& \text{iff} \quad (\widehat{\chi},\widehat{H}) \models rhs^{l_2} \ \wedge\ \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \\
\widehat{[\textit{Assign 2}]} & (\widehat{\chi},\widehat{H}) \models (\texttt{Var}^{l_o}.[C]\texttt{MemberName})^{l_1} \texttt{ = } rhs^{l_2} \\
& \text{iff} \quad (\widehat{\chi},\widehat{H}) \models \texttt{Var}^{l_o} \ \wedge\ (\widehat{\chi},\widehat{H}) \models rhs^{l_2} \ \wedge\ \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \ \wedge \\
& \qquad \forall \texttt{C}' \in \widehat{\chi}(l_o) : \widehat{\chi}(l_1) \subseteq \widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \\
\widehat{[\textit{Call}]} & (\widehat{\chi},\widehat{H}) \models (\texttt{Var}^{l_o}\texttt{.[AT]MethodName(VarName}^{l_1}\texttt{, } \cdots \texttt{, VarName}^{l_n}\texttt{))}^{l} \\
& \text{iff} \quad (\widehat{\chi},\widehat{H}) \models \texttt{Var}^{l_o} \ \wedge\ \forall 1 \leq j \leq n : (\widehat{\chi},\widehat{H}) \models \texttt{VarName}^{l_j} \ \wedge \\
& \qquad \forall \texttt{C} \in \widehat{\chi}(l_o) : \textit{MostSpecial}(\texttt{MethodName}, \texttt{C}, \textit{AT}) = \\
& \qquad\quad \{(\texttt{C}', \textit{MT}', \texttt{T}_r\ \texttt{MethodName}(\texttt{T}_1\ \texttt{x}_1^{lp_1}, \cdots, \texttt{T}_n\ \texttt{x}_n^{lp_n})\{\textit{Stmts}\}^{l_r})\} \\
& \qquad \Rightarrow \forall j = 1, \cdots, n : \widehat{\chi}(l_j) \subseteq \widehat{\chi}(lp_j) \ \wedge\ (\widehat{\chi},\widehat{H}) \models \textit{Stmts} \wedge \\
& \qquad\quad \widehat{\chi}(l_r) \subseteq \widehat{\chi}(l)
\end{array}
$$

Table 4.2: Acceptability relation $\models$ (cont.)

The $\widehat{[\textit{If}]}$ rule contains recursive references to the relation. These demand that $(\widehat{\chi},\widehat{H})$ is consistent with the two blocks $\textit{Block}_1$ and $\textit{Block}_2$. $\widehat{[\textit{Return 1}]}$ handles returning a variable. This is acceptable, if all reaching definitions of the variable are collected and flow into the label of the body of the procedure. Remember that this label has been added exactly for this purpose; namely as an interface from a called method to the callee. The remaining rules $\widehat{[\textit{Return 2}]}$ and $\widehat{[\textit{Skip}]}$ are trivial.

The rule for assignments has been split into two parts: the first handles assignments to local variables, the second to class and instance members. Both have two things in common. First, they require $(\widehat{\chi},\widehat{H})$ to be consistent with the right hand side. Second, they check that the possible values flow from the label l_2 of the right hand side to the label l_1 of the left hand side. For rule $\widehat{[\textit{Assign 1}]}$ this is all that needs to be done as for all using occurrences of the local variable VarName the label l_1 has been recorded in rd.

However, for rule $\widehat{[\textit{Assign 2}]}$ the values associated with l_1 need to flow to all possibly addressed members. These are identified by the classes that may occur at label l_o, that is all possible definitions as determined by $\models$.

Last but not least we need to handle calls. To do so, first $(\widehat{\chi},\widehat{H})$ is checked to be consistent with the object or class on that the call is based and the arguments to the call. Having done

so, the possible run-time types of the object must have been stored in $\widehat{\chi}(l_o)$. Based on this set the possible target methods are identified by means of function *MostSpecial*. As pointed out in Section 4.1, the handling of this step is one of the key points of our approach. For each callable method `m` it is checked that the labels of arguments flow to the labels of parameters. Furthermore, if the method returns a class value, the set associated with the label l_r of the method body must flow back to the set associated with the label l of the call.

4.2.3 Well-Definedness of the Acceptability Relation $\models$

Later on we will need to be able to use some properties of the just defined relation. However, we still have not justified that $\models$ is well-defined. As stated in [NNH1999] the problem is that the $\widehat{[\mathit{Call}]}$ clause from Table 4.2 is not in a form that allows checking well-definedness by structural induction. It checks whether $(\widehat{\chi}, \widehat{H})$ allows the analysis of the body of all possibly called methods. In contrast to the other rules, this implies that something *not* syntactically smaller needs to be analyzed. Instead, the method body will probably be bigger than the call statement itself. Even worse, the method could call itself recursively. Therefore, we will need to define $\models$ by coinduction as the greatest fixed point of a function.

Following [NNH1999] we view Tables 4.1 and 4.2 as defining a function

$$\kappa : ((\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Heap}} \times \mathbf{Exp}) \to \{\mathtt{true}, \mathtt{false}\}) \to ((\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Heap}} \times \mathbf{Exp}) \to \{\mathtt{true}, \mathtt{false}\})$$

The argument is a function K and the result another function. We then have e. g.

$$\kappa(K)(\widehat{\chi}, \widehat{H}, \mathtt{return\ Variable}^l) = K(\widehat{\chi}, \widehat{H}, \mathtt{Variable}^l) \ \wedge \ \widehat{\chi}(l) \subseteq \widehat{\chi}(l_r)$$

In order to apply a coinductive definition, we need to show that κ is a monotone function on the complete lattice $(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Heap}} \times \mathbf{Exp}) \to \{\mathtt{true}, \mathtt{false}\}, \sqsubseteq)$. $\sqsubseteq$ is defined by $K_1 \sqsubseteq K_2$ iff $\forall(\widehat{\chi}, \widehat{H}, e) : (K_1(\widehat{\chi}, \widehat{H}, e) = \mathtt{true}) \Rightarrow (K_2(\widehat{\chi}, \widehat{H}, e) = \mathtt{true})$.

Theorem 4.1 *Let* $M = (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Heap}} \times \mathbf{Exp}) \to \{\mathtt{true}, \mathtt{false}\}$, $L = (M, \sqsubseteq)$ *and* $\kappa : M \to M$. *Then* κ *is a monotone function on* L.

Proof Let $K_1, K_2 \in M$. We need to show that $K_1 \sqsubseteq K_2 \Rightarrow \kappa(K_1) \sqsubseteq \kappa(K_2)$. By applying the definition of $\sqsubseteq$, this equals

$$\begin{aligned}(\forall(\widehat{\chi}, \widehat{H}, t) : K_1(\widehat{\chi}, \widehat{H}, t) = \mathtt{true} \Rightarrow K_2(\widehat{\chi}, \widehat{H}, t) = \mathtt{true}) \qquad & (\star)\\ \Rightarrow \forall(\widehat{\chi}, \widehat{H}, t) : \kappa(K_1)(\widehat{\chi}, \widehat{H}, t) = \mathtt{true} \Rightarrow \kappa(K_2)(\widehat{\chi}, \widehat{H}, t) = \mathtt{true} \quad & (\star\star)\end{aligned}$$

In order to complete the proof we assume $(\star)$ and show $(\star\star)$ for every rule from Tables 4.1 and 4.2. This is rather easy, for the rules that always hold it is even trivial.

We show the step exemplarily for one rule, namely for $\widehat{[\mathit{Assign\ 1}]}$. Define $\mathtt{t} \equiv \mathtt{VarName}^{l_1}$ `=` $\mathtt{rhs}^{l_2}$. We need to show $\kappa(K_1)(\widehat{\chi}, \widehat{H}, \mathtt{t}) = \mathtt{true} \Rightarrow \kappa(K_2)(\widehat{\chi}, \widehat{H}, \mathtt{t}) = \mathtt{true}$. By the definition of κ we get

$\kappa(K_1)(\widehat{\chi}, \widehat{H}, \mathtt{t}) = K_1(\widehat{\chi}, \widehat{H}, \mathtt{t}) \ \wedge \ \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1)$

With $(\star)$ we get

$$\Rightarrow K_2(\widehat{\chi}, \widehat{H}, \mathtt{t}) \ \wedge \ \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1)$$

and with the definition of κ this yields

$$= \kappa(K_2)(\widehat{\chi}, \widehat{H}, \mathtt{t})$$

The other rules are shown analogously. ∎

From Tarski's Fixed Point Theorem follows that function κ has fixed points and as a consequence $\models$ can be defined coinductively. Namely, $\models$ is the greatest fixed point of κ.

4.2.4 Semantic Correctness

We will now close a big theoretical gap in this work, the need to ensure that the information computed by our analysis really is a safe description of the run-time behavior of the program. This means that no labeled program point may take on a class which is not in the set computed by the control flow analysis.

The analysis given above still needs to be extended to handle the intermediate expressions and statements from the semantics of $\mathtt{Bahasa}_r$. This is straightforward and is given in Table 4.3. The rule for the `bind` statement reveals why the (global) heap is passed on as an argument: we need to be able to speak about the heap when checking this rule.

In order to establish the proof and to define a relation $\models$, we will need a relation $\mathcal{T}$ between the run-time environment σ and the heap H on the one side and the abstract environment $(\widehat{\chi}, \widehat{H})$ on the other. This will be defined by means of another relation $\mathcal{V}$ that relates references to objects with sets of class names.

Definition 4.2 *(Correctness Relations)*
Let $\mathcal{T} \subseteq (\textbf{\textit{Env}} \times \textbf{\textit{Heap}}) \times (\widehat{\textbf{\textit{Cache}}} \times \widehat{\textbf{\textit{Heap}}})$ *and* $\mathcal{V} \subseteq \textbf{\textit{References}} \times \widehat{\textbf{\textit{Values}}}$.

We define

$$\iota_i^C \ \mathcal{V} \ M \quad \textit{iff} \quad C \in M$$

$$(\sigma, H) \ \mathcal{T} \ (\widehat{\chi}, \widehat{H}) \quad \textit{iff} \quad \begin{cases} \forall x \in dom(\sigma) : \sigma(x) = (l, val) \Rightarrow \forall l' \in DU(x, l) : val \ \mathcal{V} \ \widehat{\chi}(l') \\ \textit{and} \ \forall \iota_i^C \in dom(H) : H(\iota_i^C) = \ll f_1 C_1 : val_1, \cdots, \\ \qquad f_n C_n : val_n \gg^C \Rightarrow val_j \ \mathcal{V} \ \widehat{H}(C, f_j, C_j) \ \textit{for} \ 1 \leq j \leq n \end{cases}$$

The relation $\mathcal{V}$ checks that the class for an object on the heap is a member of a set of classes. This relation is used by $\mathcal{T}$ twice. The first use is to check variables. Let `x` be a variable that was

$$
\begin{array}{lllll}
\widehat{[\textit{Inter 1}]} & (\widehat{\chi},\widehat{H}) & \models & ((\iota_i^{C'})^{l_o}\,.\,\texttt{[C]MemberName})^l \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models (\iota_i^{C'})^{l_o} \wedge \widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \subseteq \widehat{\chi}(l) \\
\\
\widehat{[\textit{Inter 2}]} & (\widehat{\chi},\widehat{H}) & \models & ((\iota_i^{C'})^{l_o}\,.\,\texttt{[C]MemberName})^{l_1}\ \texttt{=}\ rhs^{l_2} \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models (\iota_i^{C'})^{l_o} \wedge (\widehat{\chi},\widehat{H}) \models rhs^{l_2} \wedge \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \wedge \\
 & & & \widehat{\chi}(l_1) \subseteq \widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \\
\\
\widehat{[\textit{Inter 3}]} & (\widehat{\chi},\widehat{H}) & \models & ((\iota_i^{C})^{l_o}\,.\,\texttt{[AT]m}(val_1^{l_1}, \cdots, val_n^{l_n})^l \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models (\iota_i^{C'})^{l_o} \wedge \forall 1 \leq j \leq n : (\widehat{\chi},\widehat{H}) \models val_j^{l_j} \wedge \\
 & & & MostSpecial(\texttt{m}, \texttt{C}, AT) = \\
 & & & \qquad \{(\texttt{C}', MT', \texttt{T}_r\ \ \texttt{m}(\texttt{T}_1\ \texttt{x}_1^{lp_1}, \cdots, \texttt{T}_n\ \texttt{x}_n^{lp_n})\{\textit{Stmts}\}^{l_r})\} \\
 & & & \wedge \forall 1 \leq j \leq n : \widehat{\chi}(l_j) \subseteq \widehat{\chi}(lp_j)\ \wedge\ (\widehat{\chi},\widehat{H}) \models \textit{Stmts} \wedge \\
 & & & \widehat{\chi}(l_r) \subseteq \widehat{\chi}(l) \\
\\
\widehat{[\textit{Inter 4}]} & (\widehat{\chi},\widehat{H}) & \models & v^l \\
 & & \text{iff} & v^l\ \mathcal{V}\ \widehat{\chi}(l) \\
\\
\widehat{[\textit{Inter 5}]} & (\widehat{\chi},\widehat{H}) & \models & (\texttt{bind}\ \sigma, H\ \texttt{in}\ \textit{Stmts})^{l_r} \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models \textit{Stmts} \wedge (\sigma, H)\ \mathcal{T}\ (\widehat{\chi},\widehat{H})
\end{array}
$$

Table 4.3: Acceptability of intermediate expressions and statements

last set at label l to point to an object with an arbitrary type $\texttt{C}$. $\mathcal{T}$ verifies that this type is member of all using labels of the definition l. The second use is to verify that for every object o on the heap all abstractions of the fields f of o contain the actual types of objects stored in f. That is, $\mathcal{V}$ and $\mathcal{T}$ state that the abstract domain is a valid description of the concrete one.

Later on we will need the following lemmata. The first one allows to replace the set on the right hand side of $\mathcal{V}$ with a superset. The second one allows to replace in $\widehat{H}(\texttt{C}_o, \texttt{f}, \texttt{C})$ the class $\texttt{C}$ with the actual class defining $\texttt{C}'$.

Lemma 4.1 *Let $v \in$ **Values** be a value and $M_1, M_2 \in \widehat{\textbf{Values}}$ be a set of class names with $v\ \mathcal{V}\ M_1$.*

$$M_1 \subseteq M_2 \Rightarrow v\ \mathcal{V}\ M_2$$

This can be easily seen as $v\ \mathcal{V}\ M_1 \Rightarrow v \in M_1 \subseteq M_2$.

Lemma 4.2 *Let* $\mathtt{C}_o, \mathtt{C}, \mathtt{C}' \in$ ***Classes*** *with* $\mathtt{C} \sqsubseteq_\Gamma \mathtt{C}'$, $\mathtt{f} \in$ ***FNames*** *and* $\mathtt{T} \in$ ***OOTypes***.

$$FieldDecl(\Gamma, \mathtt{C}, \mathtt{f}) = (\mathtt{C}', \mathtt{T}) \Rightarrow \widehat{H}(\mathtt{C}_o, \mathtt{f}, \mathtt{C}) = \widehat{H}(\mathtt{C}_o, \mathtt{f}, \mathtt{C}')$$

The basis for the proof of semantic correctness is the operational semantics from Chapter 3. The proof of correctness is a *subject reduction proof*, meaning that a solution obtained remains correct under evaluation.

The idea of the theorem is to relate the concrete and the abstract world. On the concrete side, we have a `Bahasa` term s, environments σ and σ', and heaps H and H'. On the abstract side, we have an abstract environment $\widehat{\chi}$ and an abstract heap $\widehat{H}$. The theorem assumes that s is rewritten to s' by the rules and that $(\widehat{\chi}, \widehat{H})$ abstract the concrete state and environment H and σ. The third assumption is that the relation $\models$ accepts $(\widehat{\chi}, \widehat{H})$ as a correct solution for s. The claim is, that then the state and the heap resulting from rewriting also are abstracted by $(\widehat{\chi}, \widehat{H})$ and that these also are correct regarding $\models$ for s'.

Theorem 4.2 *(Semantic Correctness)*
Let `p` *be a* $\mathtt{Bahasa}_r$ *program,* Γ *the type environment induced by* p.

If

$$\begin{array}{ll} \langle s, \sigma, H\rangle \rightsquigarrow_P \langle s', \sigma', H'\rangle & \texttt{(A1)} \\ (\sigma, H)\ \mathcal{T}\ (\widehat{\chi}, \widehat{H}) & \texttt{(A2)} \\ (\widehat{\chi}, \widehat{H}) \models \mathtt{s} & \texttt{(A3)} \end{array}$$

then holds

$$\begin{array}{ll} (\sigma', H')\ \mathcal{T}\ (\widehat{\chi}, \widehat{H}) & \texttt{(C1)} \\ (\widehat{\chi}, \widehat{H}) \models \mathtt{s'} & \texttt{(C2)} \end{array}$$

Proof We assume $(\widehat{\chi}, \widehat{H}) \models \mathtt{s}$ and prove $(\widehat{\chi}, \widehat{H}) \models \mathtt{s}'$ by induction on the inference tree for $\langle s, \sigma, H\rangle \rightsquigarrow_P \langle s', \sigma', H\rangle$. The rules for executing programs have been given in Tables 3.8, 3.9, 3.10 and 3.11 in Chapter 3. Relation $\models$ is defined in Table 4.1, 4.2 and 4.3.

We start with the base cases, i. e. those rules that do not require any premises to be fulfilled.

[Variable]
The first rule describes a using occurrence of a variable. The assumptions of the theorem and the correctness relation state

$$\begin{array}{lll} \texttt{(A1)} & \Rightarrow & \langle \mathtt{VarName}^l, \sigma, H\rangle \rightsquigarrow_P \langle \sigma(\mathtt{VarName})^l, \sigma, H\rangle \quad (1) \\ \texttt{(A2)} & \Rightarrow & \sigma(\mathtt{VarName}) = (l_1, \iota_i^C) \Rightarrow \forall l' \in DU(\mathtt{VarName}, l_1) : val\ \mathcal{V}\ \widehat{\chi}(l') \quad (2) \\ \texttt{(A3)} & \Rightarrow & \forall l'' \in rd(l) : \widehat{\chi}(l'') \subseteq \widehat{\chi}(l) \quad (3) \end{array}$$

`(C1)` follows immediately from `(A2)` as neither σ nor H change. (2) states that l_1 is the label of the left hand side of the last assignment to `VarName`. That is, $l_1 \in rd(l)$ and with (3) this results in $\widehat{\chi}(l_1) \subseteq \widehat{\chi}(l) \Leftrightarrow \iota_i^C\ \mathcal{V}\ \widehat{\chi}(l)$. With the last rule of Table 4.3 this proves `(C2)`.

[Field Access 2]
This rule describes how to evaluate a using occurrence of an access to the field of an object based on address $\iota_i^{C'}$. The assumptions are

$$\begin{aligned}
&\texttt{(A1)} \quad \Rightarrow \quad \left\langle (\iota_i^{C'\,l_o}.[C]f)^l, \sigma, H \right\rangle \rightsquigarrow_P \left\langle (getfield(H(\iota_i^{C'}), f, C))^l, \sigma, H \right\rangle && (4)\\
&\texttt{(A2)} \quad \Rightarrow \quad H(\iota_i^{C'}) = \ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C'} \\
&\qquad\qquad\qquad\qquad \Rightarrow val_j \;\mathcal{V}\; \widehat{H}(C', f_j, C_j), 1 \leq j \leq n && (5)\\
&\texttt{(A3)} \quad \Rightarrow \quad (\widehat{\chi}, \widehat{H}) \models (\iota_i^{C'\,l_o}) \wedge \forall C'' \in \widehat{H}(l_o) : \widehat{H}(C'', f, C) \subseteq \widehat{\chi}(l) && (6)
\end{aligned}$$

As σ and H are not changed, again `(C1)` follows from `(A2)`. For `(C2)` we need to show that

$$getfield(H(\iota_i^{C'}), f, C) \;\mathcal{V}\; \widehat{\chi}(l) \tag{7}$$

holds. From the definition of $getfield$ and (5) we get with $f = f_j$ for some $1 \leq j \leq n$

$$getfield(H(\iota_i^{C'}), f, C) = val_j \quad \mathcal{V} \quad \widehat{H}(C', f, C_j) \tag{8}$$

With Lemma 4.2 and the definition of $\mathcal{V}$ we get $val_j \;\mathcal{V}\; \widehat{H}(C', f_j, C)$. From (6) follows $\iota_i^{C'} \;\mathcal{V}\; \widehat{H}(l_o) \Leftrightarrow C' \in \widehat{H}(l_o) \Rightarrow val_j = \widehat{H}(C', f, C) \subseteq \widehat{H}(l)$.

[Object Creation]
The last rule from evaluation of expressions is the one for object creation. The assumptions are

$$\begin{aligned}
&\texttt{(A1)} \quad \Rightarrow \quad \left\langle \texttt{new C}, \sigma, H \right\rangle \rightsquigarrow_P \left\langle \iota_i^C, \sigma, H[\iota_i^C \mapsto \ll\gg^C] \right\rangle && (9)\\
&\texttt{(A2)} \quad \Rightarrow \quad H(\iota_i^{C'}) = \ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C'} \\
&\qquad\qquad\qquad\qquad \Rightarrow val_j \;\mathcal{V}\; \widehat{H}(C', f_j, C_j), 1 \leq j \leq n && (10)\\
&\texttt{(A3)} \quad \Rightarrow \quad \{C\} \subseteq \widehat{\chi}(l) && (11)
\end{aligned}$$

As σ is unchanged during evaluation, in order to show `(C1)` we need to show

$$\begin{aligned}
&\forall \iota_j^{C''} \in dom(H') : H'(\iota_j^{C''}) = \ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C'} \\
&\qquad\qquad \Rightarrow val_j \;\mathcal{V}\; \widehat{H}(C'', f_j, C_j), 1 \leq j \leq n && (12)
\end{aligned}$$

For all $\iota_j^{C''} \neq \iota_i^C$ this holds by (10). For the newly introduced ι_i^C in (9) note that all fields in the created object are initialized with the default value of their type, that is `null` for class and interface types. However, we then have $val_k \;\mathcal{V}\; \widehat{H}(C_k, f_k)$, so (12) holds.

In order to show $(\widehat{\chi}, \widehat{H}) \models (\iota_i^C)^l$ we must show

$$\iota_i^C \;\mathcal{V}\; \widehat{\chi}(l) \tag{13}$$

By definition of $\mathcal{V}$ this is equal to showing $C \in \widehat{\chi}(l)$. With (11) we have $\{C\} \subseteq \widehat{\chi}(l)$, so (13) holds.

[If then else 1,2], [Return 2], [Skip]
Trivial, as they change neither state nor heap and leave the term part almost unchanged.

[Assign 3]
From the assignment rule we get $\sigma' = \sigma[\texttt{VarName} \mapsto (l_1, \texttt{val})]$

$$\texttt{(A1)} \Rightarrow \left\langle \texttt{VarName}^{l_1} \texttt{ = val}^{l_2}, \sigma, H \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H \right\rangle \quad (14)$$

$$\texttt{(A2)} \Rightarrow \forall x \in dom(\sigma) : \sigma(x) = (l, v) \Rightarrow \forall l' \in DU(x, l) : v \; \mathcal{V} \; \widehat{\chi}(l') \quad (15)$$

$$\texttt{(A3)} \Rightarrow (\widehat{\chi}, \widehat{H}) \models \texttt{val}^{l_2} \wedge \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \quad (16)$$

As $(\widehat{\chi}, \widehat{H}) \models \epsilon$ we obtain (C2) and only need to show $(\sigma', H) \; \mathcal{T} \; (\widehat{\chi}, \widehat{H})$. However, as H does not change we only need to prove

$$\forall y \in dom(\sigma') : \sigma'(y) = (l, v) \Rightarrow \forall l' \in DU(y, l) : v \; \mathcal{V} \; \widehat{\chi}(l') \quad (17)$$

There are two cases. First, if $y \neq \texttt{VarName}$ then we get (17) from (A2) because then $\sigma'(y) = \sigma(y)$.

Assume $y = \texttt{VarName}$. From the premises follows $\sigma'(\texttt{VarName}) = (l_1, \texttt{val})$, so instead of (17) we need to show

$$\forall l' \in DU(\texttt{VarName}, l_1) : \texttt{val} \; \mathcal{V} \; \widehat{\chi}(l'') \quad (18)$$

From (A3) and Lemma 4.1 we get

$$(\widehat{\chi}, \widehat{H}) \models \texttt{val}^{l_2} \Leftrightarrow \texttt{val}^{l_2} \; \mathcal{V} \; \widehat{\chi}(l_2) \Rightarrow \texttt{val}^{l_2} \; \mathcal{V} \; \widehat{\chi}(l_1) \quad (19)$$

Every using occurrence of the value assigned to VarName at label l_1 will be of the form $\texttt{VarName}^{l_u}$. From the definition of $\models$ we get

$$\forall l_2 \in rd(l_u) : \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_u) \quad (20)$$

However, l_1 is the label of a definition reaching l_u, so from (20) we get $\widehat{\chi}(l_1) \subseteq \widehat{\chi}(l_u)$ and with (19) this results in $\texttt{val}^{l_2} \; \mathcal{V} \; \widehat{\chi}(l_u)$ for all $l_u \in DU(\texttt{VarName}, l_1)$.

[Assign 4]
With premises $H(\iota_i^{C'}) = \ll \cdots \gg^{C'}$ and $H' = H[\iota_i^{C'} \mapsto setfield(H(\iota_i^{C'}, f, C, val)]$ the assumptions are

$$\begin{aligned}
\texttt{(A1)} \quad &\Rightarrow \quad \left\langle ((\iota_i^{C'})^{l_o}\texttt{.[C]f})^{l_1} \texttt{ = val}^{l_2}, \sigma, H \right\rangle \rightsquigarrow_P \left\langle , \sigma, H' \right\rangle && (21)\\
\texttt{(A2)} \quad &\Rightarrow \quad H(\iota_j^{C''}) =\ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C''} \\
&\qquad\qquad \Rightarrow val_j \; \mathcal{V} \; \widehat{H}(C'', f_j, C_j), 1 \leq j \leq n && (22)\\
\texttt{(A3)} \quad &\Rightarrow \quad (\widehat{\chi}, \widehat{H}) \models ((\iota_i^{C'})^{l_o}\texttt{.f})^{l_1} \texttt{ = val}^{l_2} \\
&\Leftrightarrow \quad (\widehat{\chi}, \widehat{H}) \models (\iota_i^{C'})^{l_o} \wedge (\widehat{\chi}, \widehat{H}) \models val^{l_2} \wedge \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \wedge \\
&\qquad \forall C_o \in \widehat{\chi}(l_o) : \widehat{\chi}(l_1) \subseteq \widehat{H}(C_o, f, C) && (23)
\end{aligned}$$

As before, `(C2)` is trivially fulfilled. As σ is unchanged, all we need to show is

$$\forall \iota_l^{C''} \in dom(H') : H'(\iota_l^{C''}) =\ll \cdots \gg^{C''} \Rightarrow val_k \; \mathcal{V} \; \widehat{H}(C'', f_k, C_k), 1 \leq k \leq n \qquad (24)$$

with $H' = H[\iota_i^{C'} \mapsto setfield(H(\iota_i^{C'}), f, C, val)]$.

There are two cases to distinguish. First assume $\iota_l^{C''} \neq \iota_i^{C'}$. Assume $o = H(\iota_l^{C''}) =\ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C''}$ and $o' = H'(\iota_l^{C''}) =\ll f_1' C_1' : val_1', \cdots, f_n' C_n' : val_n' \gg^{C''}$. As only the object stored at ι_i^C is changed we have $\forall 1 \leq j \leq n : val_j = val_j'$.

The second case is $\iota_l^{C''} = \iota_i^{C'}$. Assume o and o' to represent the objects stored in H respectively H', i. e. $o = H(\iota_l^{C''}) =\ll f_1 C_1 : val_1, \cdots, f_n C_n : val_n \gg^{C''}$ and $o' = H'(\iota_l^{C''}) =\ll f_1' C_1' : val_1', \cdots, f_n' C_n' : val_n' \gg^{C''}$.

We now need to inspect all fields f_j in the objects. We fix f_k to be the field that has been changed by $setfield$. Again there are two cases. If $f_k \neq f_j$ then val_j is the same in o and o', so (24) holds because of (22).

If $f_k = f_j$ then $val_j = val$ by definition of $setfield$. We need to show

$$val \; \mathcal{V} \; \widehat{H}(C', f_j, C_j) \qquad (25)$$

For the only interesting case $val = \iota_v^{C_v}$ we get from (23) that

$$\begin{aligned}
(\widehat{\chi}, \widehat{H}) \models (\iota_i^{C'})^{l_o} \quad &\Leftrightarrow \quad C' \in \widehat{\chi}(l_o) && (26)\\
(\widehat{\chi}, \widehat{H}) \models val^{l_2} \quad &\Leftrightarrow \quad val \; \mathcal{V} \; \widehat{\chi}(l_2) && (27)
\end{aligned}$$

From Lemma 4.1, (23) and (27) follows that $val \; \mathcal{V} \; \widehat{\chi}(l_1)$. From (23) and (26) we get $\widehat{\chi}(l_1) \subseteq \widehat{H}(C', f, C)$ and with Lemma 4.2 this yields $val \; \mathcal{V} \; \widehat{H}(C', f, C) = \widehat{H}(C', f, C_j)$.

[Call 2,3]
Calls to instance and class methods are almost identical with the only difference in method lookup. We will describe the case for instance methods. Class methods are handled analogously.

For a call to an instance method the important premise is $MostSpecial(m, C, AT) = (C', MT', T_r \;\; \texttt{m}(T_1 \; x_1^{lp_1}, \cdots, T_n \; x_n^{lp_n})\{ \;\; \textit{Stmts} \;\; \})$ that states how to identify the method called. The assumptions made by the theorem are

$$\texttt{(A1)} \;\Rightarrow\; \left\langle (\iota_i^C)^{l_o}\,.\,\texttt{[AT]m(val}_1^{l_1},\cdots,\texttt{val}_n^{l_n}\texttt{)}, \sigma, H\right\rangle \rightsquigarrow_P \left\langle \texttt{bind } \sigma', H \texttt{ in } \textit{Stmts}\texttt{; end}, \sigma, H\right\rangle \quad (28)$$

$$\texttt{(A3)} \;\Rightarrow\; (\widehat{\chi}, \widehat{H}) \models (\iota_i^C)^{l_o}\,.\,\texttt{[AT]m(val}_1^{l_1},\cdots,\texttt{val}_n^{l_n}\texttt{)} \quad (29)$$

We do not need to show anything for `(C1)` as neither σ nor H are changed. However, what needs to be proven is

$$(\widehat{\chi}, \widehat{H}) \models \texttt{bind } \sigma', H \texttt{ in } \textit{Stmts} \;\Leftrightarrow\; (\widehat{\chi}, \widehat{H}) \models \textit{Stmts} \wedge (\sigma', H)\;\mathcal{T}\;(\widehat{\chi}, \widehat{H}) \quad (30)$$

As stated before, H is left unchanged, so we only need to handle σ' and $\widehat{\chi}$ in (30). From the premises of the operational semantics we get

$$\sigma' \;=\; \sigma[x_1 \mapsto (l_1, val_1), \cdots, x_n \mapsto (l_n, val_n)] \quad (31)$$

With (29) and Lemma 4.1 we get $val_k\;\mathcal{V}\;\widehat{\chi}(lp_k)$. As the parameters are definition points it follows by the definition of DU that the values (l_k, val_k) will flow to all uses. This completes the proof for this case.

Having completed the handling of the base cases we will now prove the induction step. Most of the still missing rules are just rewrite rules in the sense that they rewrite expressions ([If then else 3, Return 1, Assign 2, Call 1, Field Access 1, Assign 1]) to ground or l-ground expressions. Those rules substitute one expression by another one and are trivially true. The same holds for the rules ([Sequence 1, Sequence 2, Bind 1]) that separate statements from blocks and execute them. We will exemplarily prove the rule for returning from a method call.

[Bind 3]
This rule is taking care of returning a value from a called to the calling procedure. From the theorem we get

$$\texttt{(A1)} \;\Rightarrow\; \left\langle (\texttt{bind } \sigma', H' \texttt{ in return val}^{l}\texttt{;s';end})^{l_r}, \sigma, H\right\rangle \rightsquigarrow_P \left\langle (\texttt{val'})^{l_r}, \sigma, H'\right\rangle \quad (32)$$

$$\texttt{(A2)} \;\Rightarrow\; (\sigma, H)\;\mathcal{T}\;(\widehat{\chi}, \widehat{H}) \quad (33)$$

$$\texttt{(A3)} \;\Rightarrow\; (\widehat{\chi}, \widehat{H}) \models \texttt{bind } \sigma', H' \texttt{ in return val}^{l} \wedge (\sigma', H')\;\mathcal{T}\;(\widehat{\chi}, \widehat{H}) \quad (34)$$

and need to show

$$(\sigma, H')\;\mathcal{T}\;(\widehat{\chi}, \widehat{H}) \quad (35)$$

$$(\widehat{\chi}, \widehat{H}) \models (\texttt{val'})^{l_r} \quad (36)$$

(35) follows immediately from (34) for H' and from (33) for σ. To show (36) we will need the induction hypothesis. The precondition of rule [bind 3] is $\left\langle (\texttt{val})^{l}, \sigma', H'\right\rangle \rightsquigarrow_P \left\langle (\texttt{val'})^{l_r}, \sigma', H'\right\rangle$.

assume $\mathtt{p} \equiv \mathtt{p'}\ \mathtt{class\ C}\ \{\ \mathit{MethDecl}^*\ \}\ \mathtt{p''}$

$\widehat{[\mathit{Program}]}_{\mathcal{S}}$	$(\widehat{\chi}, \widehat{H})$	$\models_S$	p
		iff	$(\widehat{\chi}, \widehat{H}) \models_S \mathtt{class\ C}\ \{\ \mathit{MethDecl}^*\ \} \wedge$
			$(\widehat{\chi}, \widehat{H}) \models_S \mathtt{p'}$
$\widehat{[\mathit{Class}]}_{\mathcal{S}}$	$(\widehat{\chi}, \widehat{H})$	$\models_S$	$\mathtt{class\ C}\ \{\ \mathit{MethDecls}\ \}$
		iff	$(\widehat{\chi}, \widehat{H}) \models_S \mathit{MethDecls}^*$
$\widehat{[\mathit{MethodDecls}]}_{\mathcal{S}}$	$(\widehat{\chi}, \widehat{H})$	$\models_S$	$\mathit{Method}\ \ \mathit{MethodDecls}$
		iff	$(\widehat{\chi}, \widehat{H}) \models_S \mathit{Method} \wedge (\widehat{\chi}, \widehat{H}) \models_S \mathit{MethDecls}$
$\widehat{[\mathit{Method}]}_{\mathcal{S}}$	$(\widehat{\chi}, \widehat{H})$	$\models_S$	$\mathit{Mod}\ \ \mathit{RetType}\ \mathtt{MethodId}\ \{\ \mathit{Stmts}\ \}\ \mathit{MethDecls}$
		iff	$(\widehat{\chi}, \widehat{H}) \models_S \mathit{Stmts} \wedge (\widehat{\chi}, \widehat{H}) \models_S \mathit{MethDecls}$

Table 4.4: Rules for analyzing whole programs

The other assumptions to be fulfilled are $(\sigma', H')\ \mathcal{T}\ (\widehat{\chi}, \widehat{H})$ and $(\widehat{\chi}, \widehat{H}) \models (\mathtt{val})^l$. The first has just been shown, the second immediately follows from the definition of $\models$ for the intermediate statement `bind` in (34) due to

$$(\widehat{\chi}, \widehat{H}) \models \mathtt{return\ val}^l \ \Leftrightarrow\ (\widehat{\chi}, \widehat{H}) \models \mathtt{val}^l \wedge \widehat{\chi}(l) \subseteq \widehat{\chi}(l_r) \tag{37}$$

Thus we get from the hypothesis

$$(\widehat{\chi}, \widehat{H}) \models (\mathtt{val'})^l \tag{38}$$

and with (37) we have $val'\ \mathcal{V}\ \widehat{\chi}(l) \subseteq \widehat{\chi}(l_r)$. With Lemma 4.1 this completes the proof. ∎

4.3 Syntax Directed Control Flow Analysis

Having proven the semantic correctness of the abstract analysis we will now turn to implementing it. However, up to now a method is analyzed at every call site. This property has been necessary for the proof of correctness but of course is undesirable for an implementation of a 0-CFA. There, one would like to analyze the whole program in one run. This is the main target in a reformulation of $\models$ to $\models_S$.

The rules are given in Tables 4.5 and 4.6; they are mostly equivalent to those from Tables 4.1 and 4.2. Actually the only difference – however an important one – is in the last rule. When calling a method we do not enforce that the body of the method called is analyzed.

We now may omit intermediate expressions and statements introduced in the previous section. Those only helped in proving semantic correctness. In contrast to the abstract analysis, we now start with analyzing the whole program and visiting every method in every class. Each method is analyzed exactly once. The rules for initiating the analysis for a program p are given in Table 4.4. We will define $\models_S$ to be the relation satisfying the specification presented. By structural induction on e one can show that there exists exactly one relation with this property. In order to facilitate the proof of Theorem 4.3 we define the relation $\models_S \equiv \mathit{gfp}(\kappa_S)$ where κ_S is the function defined in Tables 4.5 and 4.6. This is an approach similar to the one used in Section 4.2 when defining the relation $\models$.

What we will need to prove is that a solution to the syntax directed analysis will always be a solution to the abstract analysis from the previous section. We restrict the solution of our analysis to those labels, classes and field names that do actually appear in a fixed program p. Assume $\mathbb{N}_p$, $\mathbf{Classes}_p$ and $\mathbf{FNames}_p$ to be the sets of labels, class names and field names that appear in p, respectively. We define a maximal solution $(\widehat{\chi}_\star, \widehat{H}_\star)$ to be

$$\begin{array}{rcl} \widehat{\chi}_\star(l) & = & \begin{cases} \emptyset & \text{if } l \notin \mathbb{N}_p \\ \mathbf{Classes} & \text{else} \end{cases} \\ \widehat{H}_\star(C, f, C') & = & \begin{cases} \emptyset & \text{if } C, C' \notin \mathbf{Classes}_p \text{ or } f \notin \mathbf{FNames}_p \\ \mathbf{Classes} & \text{else} \end{cases} \end{array}$$

This makes the domains finite. By inspecting the rules for $\models_S$ one can see that a solution $(\widehat{\chi}, \widehat{H})$ will always fulfill $(\widehat{\chi}, \widehat{H}) \sqsubseteq (\widehat{\chi}_\star, \widehat{H}_\star)$. This will allow us to relate a solution of $\models_S$ to a solution of $\models$.

Theorem 4.3 *(Preservation of Solutions)*
If $(\widehat{\chi}, \widehat{H}) \models_S t$ *and* $(\widehat{\chi}, \widehat{H}) \sqsubseteq (\widehat{\chi}_\star, \widehat{H}_\star)$ *then* $(\widehat{\chi}, \widehat{H}) \models t$.

The proof structure is identical to a similar proof in [NNH1999]. It is based on coinduction – this approach is necessary because we had to define $\models$ coinductively in Section 4.2.3 due to the effects of method calls.

Proof Let $(\widehat{\chi}, \widehat{H}) \models_S e$ and $(\widehat{\chi}, \widehat{H}) \sqsubseteq (\widehat{\chi}_\star, \widehat{H}_\star)$. If Term_t is the set of all sub-terms of t then we get

$$\forall t' \in \mathit{Term}_t : (\widehat{\chi}, \widehat{H}) \models_S \texttt{t'} \tag{39}$$

This is obvious from the fact that the specification of $\models_S$ directly relates to the syntax of `Bahasa` programs.

$\widehat{[\mathit{Constant}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	`c`
			always
$\widehat{[\mathit{VarName}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	$\texttt{VarName}^l$
		iff	$\forall l' \in rd(l) : \widehat{\chi}(l') \subseteq \widehat{\chi}(l)$
$\widehat{[\mathit{ClassName}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	$\texttt{ClassName}^l$
		iff	$\texttt{ClassName} \in \widehat{\chi}(l)$
$\widehat{[\mathit{Member}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	$(\texttt{Var}^{l_o}\texttt{.[C]MemberName})^l$
		iff	$(\widehat{\chi},\widehat{H}) \models_S \texttt{Var}^{l_o} \wedge \forall \texttt{C}' \in \widehat{\chi}(l_o) :$
			$\widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \subseteq \widehat{\chi}(l)$
$\widehat{[\mathit{Create}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	$(\texttt{new ClassName}()\ )^l$
		iff	$\texttt{ClassName} \in \widehat{\chi}(l)$
$\widehat{[\mathit{If}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	`if` *Expr* `then` Block_1 `else` Block_2
		iff	$(\widehat{\chi},\widehat{H}) \models_S \mathit{Block}_1 \wedge (\widehat{\chi},\widehat{H}) \models_S \mathit{Block}_2$
$\widehat{[\mathit{Return\ 1}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	$\texttt{return Variable}^l$
		iff	$(\widehat{\chi},\widehat{H}) \models_S \texttt{Variable}^l \wedge \widehat{\chi}(l) \subseteq \widehat{\chi}(l_r)$
			where l_r is the label of the method body analyzed
$\widehat{[\mathit{Return\ 2}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	`return`
			always
$\widehat{[\mathit{Skip}]}_{\mathcal{S}}$	$(\widehat{\chi},\widehat{H})$	$\models_S$	`skip`
			always

Table 4.5: Syntax directed Control Flow Analysis

$$
\begin{array}{lllll}
[\widehat{\textit{Assign 1}}]_{\mathcal{S}} & (\widehat{\chi},\widehat{H}) & \models_S & \texttt{VarName}^{l_1}\ \texttt{=}\ \textit{rhs}^{l_2} \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models_S \textit{rhs}^{l_2} \wedge \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \\
[\widehat{\textit{Assign 2}}]_{\mathcal{S}} & (\widehat{\chi},\widehat{H}) & \models_S & (\texttt{Var}^{l_o}\texttt{[C]}\texttt{.MemberName})^{l_1}\ \texttt{=}\ \textit{rhs}^{l_2} \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models_S \texttt{Var}^{l_o} \wedge (\widehat{\chi},\widehat{H}) \models_S \textit{rhs}^{l_2} \wedge \widehat{\chi}(l_2) \subseteq \widehat{\chi}(l_1) \wedge \\
 & & & \quad \forall \texttt{C}' \in \widehat{\chi}(l_o) : \widehat{\chi}(l_1) \subseteq \widehat{H}(\texttt{C}', \texttt{MemberName}, \texttt{C}) \\
[\widehat{\textit{Call}}]_{\mathcal{S}} & (\widehat{\chi},\widehat{H}) & \models_S & (\texttt{Var}^{l_o}\texttt{.[AT]m(Var}^{l_1}\texttt{, } \cdots \texttt{, Var}^{l_n}\texttt{))}^{l} \\
 & & \text{iff} & (\widehat{\chi},\widehat{H}) \models_S \texttt{Var}^{l_o} \wedge \forall 1 \leq i \leq n : (\widehat{\chi},\widehat{H}) \models_S \texttt{Var}^{l_i} \wedge \\
 & & & \quad \forall \texttt{C} \in \widehat{\chi}(l_o) : \text{if } \textit{MostSpecial}(\texttt{m}, \texttt{C}, \texttt{AT}) = \\
 & & & \quad \{(\texttt{C}', \textit{MT}', \texttt{T}_r\texttt{m}(\texttt{T}_1\ \texttt{x}_1^{lp_1}, \cdots, \texttt{T}_n\ \texttt{x}_n^{lp_n})\{\textit{Stmts}\}^{l_r})\} \\
 & & & \quad \text{then } \forall 1 \leq i \leq n : \widehat{\chi}(l_i) \subseteq \widehat{\chi}(lp_i) \wedge \widehat{\chi}(l_r) \subseteq \widehat{\chi}(l)
\end{array}
$$

Table 4.6: Syntax directed Control Flow Analysis (cont.)

From the definition of the abstract and syntax directed analyses we get $\models$ to be $\mathit{gfp}(\kappa)$ and $\models_S$ to be $\mathit{gfp}(\kappa_S)$. Additionally, we define a third relation $(\widehat{\chi}', \widehat{H}') \models^\star t'$ as $(\widehat{\chi}', \widehat{H}') = (\widehat{\chi}, \widehat{H}) \wedge t' \in \mathit{Term}_t$.

If we can show that

$$(\kappa_S(\models_S) \cap \models^\star) \subseteq \kappa(\models_S \cap \models^\star) \tag{40}$$

then we get with $\kappa_S(\models_S) = \models_S$ that $(\models_S \cap \models^\star) \subseteq \kappa(\models_S \cap \models^\star)$. As κ is the greatest fixed point this results in $\models_S \cap \models^\star \subseteq \models$. Furthermore, as $t \in \mathit{Term}_t$ we have $(\widehat{\chi},\widehat{H}) \models^\star t$ and $(\widehat{\chi},\widehat{H}) \models_S \texttt{t}$ from the preconditions. However, this would give us the required $(\widehat{\chi},\widehat{H}) \models \texttt{e}$.

In order to prove (40) we will assume for every rule from Tables 4.5 and 4.6 that the right hand side holds for $(\widehat{\chi},\widehat{H},t)$ for $t \in \mathit{Term}_\star$. We then show that for the dual rule from the abstract analysis in Tables 4.1 and 4.2 the right hand holds, too, after replacing $\models$ by $\models_S \cap \models_\star$.

For all but the last rule from Table 4.6 this is obvious as they are the same in both relations. For the method call all we need to show is

$$(\widehat{\chi},\widehat{H}) \models_S \textit{Stmts} \tag{41}$$

However, *Stmts* must be a sub-term of the program t being analyzed. And as we have $(\widehat{\chi},\widehat{H}) \sqsubseteq (\widehat{\chi}_\star,\widehat{H}_\star)$ we get (41) from (39). ∎

Having proven that the syntax directed analysis is a conservative approximation of the abstract one, there only remains one more question to be answered. Namely, does the analysis really have a solution? Actually, it has. This can be derived from the following theorem that states that the set of solutions for an analysis is a Moore family. As before we now restrict $\widehat{\mathbf{Cache}}$ and $\widehat{\mathbf{Heap}}$ to only range over labels, class names, and field names occurring in p:

- $\widehat{\mathbf{Cache}}_p : \mathbb{N}_p \to \mathcal{P}(\mathbf{Classes}_p)$
- $\widehat{\mathbf{Heap}}_p : \mathbf{Classes}_p \times \mathbf{FNames}_p \times \mathbf{Classes}_p \to \mathcal{P}(\mathbf{Classes}_p)$

Theorem 4.4 *(Existence of Solutions)*
Let p be a `Bahasa` *program.*
The set $M := \{(\widehat{\chi}, \widehat{H}) \in \widehat{\boldsymbol{Cache}}_p \times \widehat{\boldsymbol{Heap}}_p | (\widehat{\chi}, \widehat{H}) \models_S \mathtt{p}\}$ is a Moore family.

Proof Using the above defined maximal element $(\widehat{\chi}_\star, \widehat{H}_\star)$ it is obvious to prove for all sub-terms t of p by structural induction on t

$$(\widehat{\chi}_\star, \widehat{H}_\star) \models_S t$$
$$\text{if } (\widehat{\chi}_1, \widehat{H}_1), (\widehat{\chi}_2, \widehat{H}_2) \in M \text{ then } ((\widehat{\chi}_1, \widehat{H}_1) \sqcap (\widehat{\chi}_2, \widehat{H}_2)) \in M$$

Thus both claims also hold for p, showing that M is not empty.

Additionally, we need to show that for every set $S \subseteq M$ the least upper bound $\sqcap S$ is in M. We note that S is finite because $\widehat{\mathbf{Cache}}_p \times \widehat{\mathbf{Heap}}_p$ is, too. By induction on the number of elements of S we can show that $\sqcap S \in M$. ∎

4.4 Generating Constraints

Finally, we can now turn to generating the basis for computing a solution to our analysis. As already stated this step is *constraint-based*. That is, the semantic directed control flow analysis generates the constraints in one run over the program. Thus the generation of constraints is quite similar to the conditions for acceptability. The algorithm for solving those will be given in the next chapter.

We start by defining the domain for constraints. As pointed out in Section 4.1, the constraints generated play a central role in making our analysis scalable.

Definition 4.3 ***(Constraints)***
The domain ***Constraints*** *is defined as the union of 3 sub-domains:*

$$\begin{aligned} \boldsymbol{Constraints} \quad = \quad & (\mathbb{N} \times \mathbb{N}) \cup \\ & (\{\mathit{LOAD}, \mathit{STORE}\} \times \boldsymbol{Members} \times \boldsymbol{Classes} \times \mathbb{N}) \cup \\ & (\{\mathit{CALL}\} \times \boldsymbol{MNames} \times \mathbb{N} \times (\mathbb{N}_\epsilon \times \mathbb{N}_\epsilon^{n \geq 0}) \end{aligned}$$

The first sub-domain describes a subset relation between two labeled points.

$[\widehat{Program}]_{\mathcal{C}}$	$C_\star[Program] := \bigcup C_\star[\texttt{C}]$ for all classes `C`defined in the program
$[\widehat{ClassBody}]_{\mathcal{C}}$	$C_\star[ClassBody] := \bigcup C_\star[\texttt{m}]$ for all methods `m` defined in class
$[\widehat{MethodDefinition}]_{\mathcal{C}}$	$C_\star[$*Mod RType* `MethodId(`*VarType* $\texttt{ParamId}_1^{l_1}$ `,` $\cdots$ `,` *VarType* $\texttt{ParamId}_n^{l_n}$`){` *Stmts*; `}`$^{l_r}] := C_\star[$*Stmts*$]$
	$labels(signature(m)) := (l_r, [l_1, \cdots, l_n])$
$[\widehat{Block}]_{\mathcal{C}}$	$C_\star[\{$*Stmts*$\}] := C_\star[$*Stmts*$]$
$[\widehat{Stmts}]_{\mathcal{C}}$	$C_\star[$*Stmt*; *Stmts*$] := C_\star[$*Stmts*$] \cup C_\star[$*Stmt*$]$
	$C_\star[\ \epsilon\ ] := \emptyset$
$\widehat{[If]}_{\mathcal{C}}$	$C_\star[$if *Expr* then *Block*$_1$ else *Block*$_2$$] := C_\star[$*Block*$_1] \cup C_\star[$*Block*$_2]$
$[\widehat{Assignment}]_{\mathcal{C}}$	$C_\star[$*Variable*^{l_1} = *rhs*^{l_2}$] := C_\star^L[$*Variable*$, l_1] \cup$
	$C_\star^R[$*rhs*$, l_2] \cup \{C(l_2) \subseteq C(l_1)\}$
$\widehat{[Call]}_{\mathcal{C}}$	$C_\star[\texttt{VarName}^l\texttt{.MethodName(VarName}_1^{l_1}\texttt{, } \cdots \texttt{, VarName}_n^{l_n}\texttt{)}] :=$
	$C_\star^R[\texttt{VarName}^l\texttt{.MethodName(VarName}\ {}_1^{l_1}\texttt{, } \cdots \texttt{, VarName}_n^{l_n}\texttt{)}, \epsilon]$
$[\widehat{Return}]_{\mathcal{C}}$	$C_\star[\texttt{return VarName}^l] := C_\star^R[\texttt{VarName}, l] \cup \{C(l) \subseteq C(l_r)\}$
	where l_r is the label of the method body analyzed
$\widehat{[Skip]}_{\mathcal{C}}$	$C_\star[\texttt{skip}] := \emptyset$

Table 4.7: Generating constraints for programs and statements

4.4.1 Access to Class Members

The second sub-domain describes access to class members. The components represent

- the name of the member,
- the static type of the variable that is used to access it,
- the label of the variable, and
- the label where the value to store (read) is taken from (written to).

By comparing with Tables 4.5 and 4.6 it becomes obvious how these components relate to the CFA. As already explained, the static type is needed in order to determine which copy of a member to access. We will shortly explain the effect of the functors.

$[\widehat{\textit{Variable}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{Variable}, l] := \bigcup_{l' \in rd(l)} \{C(l') \subseteq C(l)\}$$
$$C^L_\star[Variable, l] := \emptyset$$

$[\widehat{\textit{Instance Member}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{VarName}^{l_m}\texttt{.[C]MemberName}, l] :=$$
$$\{\texttt{LOAD}\ (\texttt{MemberName}, \texttt{C}, l_m, l)\}$$
$$\cup\, C^R_\star[\texttt{VarName}, l_m]$$
$$C^L_\star[\texttt{VarName}^{l_m}\texttt{.[C]MemberName}, l] :=$$
$$\{\texttt{STORE}\ (\texttt{MemberName}, \texttt{C}, l_m, l)\}$$
$$\cup\, C^R_\star[\texttt{VarName}, l_m]$$

$[\widehat{\textit{Class Member}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{ClassName}^{l_m}\texttt{.[C]MemberName}, l] :=$$
$$\{\texttt{LOAD}\ (\texttt{MemberName}, \texttt{ClassName}, l_m, l)\}$$
$$\cup\, \{\{\texttt{ClassName}\} \subseteq C(l_m)\}$$
$$C^L_\star[\texttt{ClassName}^{l_m}\texttt{.[C]MemberName}, l] :=$$
$$\{\texttt{STORE}\ (\texttt{MemberName}, \texttt{ClassName}, l_m, l)\}$$
$$\cup\, \{\{\texttt{ClassName}\} \subseteq C(l_m)\}$$

$[\widehat{\textit{Call}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{VarName}^{l}\texttt{.[AT]MethodName(VarName}_1^{l_1}, \cdots, \texttt{VarName}_n^{l_n}\texttt{)}, l_t] :=$$
$$C^R_\star[\texttt{VarName}, l] \cup \bigcup_{i=1\cdots n} C^R_\star[\texttt{VarName}_i, l_i]$$
$$\cup\{\texttt{CALL}\ (\texttt{MethodName}, l, l_t, [l_1, \cdots, l_n], \Gamma(\texttt{VarName}), \texttt{AT})\}$$

$[\widehat{\textit{Static}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{ClassName}^{l}\texttt{.MethodName(VarName}_1^{l_1}, \cdots, \texttt{VarName}_n^{l_n}\texttt{)}, l_t] :=$$
$$\{\{\texttt{ClassName}\} \subseteq C(l)\} \cup \bigcup_{i=1\cdots n} C^R_\star[\texttt{VarName}_i, l_i]$$
$$\cup\{\texttt{CALL}\ (\texttt{MethodName}, l, l_t, [l_1, \cdots, l_n], \texttt{ClassName}, \texttt{AT})\}$$

$[\widehat{\textit{ClassCreation}}]_{\mathcal{C}}$

$$C^R_\star[\texttt{new ClassName()}, l] := \{\{\texttt{ClassName}\} \subseteq C(l)\}$$

Table 4.8: Generating constraints for expressions

Example 4.1 *First assume a term* $t \equiv \mathtt{v}^{l_1}\,.\,\mathtt{[C]f}^{l_2}$. *When applying* $C_\star^L$ *to* t *we get the constraint* $\mathit{STORE}\,(\mathtt{f}, \mathtt{C}, l_1, l_2)$ *stating that in order to access the field* `f` *that has been defined in class* `C` *we need to look up the possible types stored in label* l_1*, namely the label of the variable* `v`*. The value to store in that field is the set associated with label* l_2*.*

Equivalently, by applying $C_\star^R$ *to* t*, we get the constraint* $\mathit{LOAD}\,(\mathtt{f}, \mathtt{C}, l_1, l_2)$*. Again, the meaning is to access field* `f` *that has been defined in class* `C` *based on the possible run-time types stored in label* l_1*. The result shall be stored in label* l_2*.*

4.4.2 Function Calls

The same holds for the third sub-domain that is generated for dynamically dispatched methods. It models the process of determining the matching method at a call site. To do so it gets several arguments:

- the method name,
- the label of the object,
- the label where a return value is written to (or ϵ if the return type is not a class type),
- the list of the labels of arguments,
- the static type of the object, and
- the tuple containing the static types of the arguments.

Again, this information will allow the solver to model the run-time behavior of the program analyzed.

Example 4.2 *Assume a statement* $s \equiv \mathtt{x}^{l_7}\,.\,\mathtt{[]m}(\mathtt{x}^{l_8})$*. The object on that* `m` *is called is sent as the* `this` *argument to method* `m`*. The result of* $C_\star$ *will contain the functor* $\mathit{CALL}\,(\mathtt{m}, l_7, \epsilon, [l_8], \mathtt{C}, [])$*. It states that based on the objects associated with the label* l_7 *the method* `m` *shall be called. We do not expect a return value but have one argument to pass on. The label where the information of that argument is stored is* l_8*. The declared type of the variable on that we call* `m` *is* `C`*. The argument type of* `m` *is empty as the* `this` *argument is ignored.*

Assume that there will be only one possible target method `tm` *with* $labels(\mathtt{tm}) = (l_{11}, [l_{10}])$*, stating that the label of the body of* `tm` *is* l_{11}*. This label will collect all possible return types of* `tm`*. Additionally,* $labels(\mathtt{tm})$ *states that the label of the argument is* l_{10}*. What will happen is that the functor will cause additional constraints to be added, namely* $\widehat{\chi}(l_8) \subseteq \widehat{\chi}(l_{10})$ *from argument to parameter label. As we do not expect a return value,* l_{11} *is ignored.*

As pointed out in Section 4.1, this modeling of the run-time behavior of the program is essential. The usual way of handling, e. g., dynamic dispatch is by means of conditional constraints [NNH1999]. These have the form $\mathtt{A} \in \widehat{\chi}(l_o) \Rightarrow \widehat{\chi}(l_1) \subseteq \widehat{\chi}(l_2)$. Using conditional constraints, a method call is translated into a set of constraints. For every method that might be called at run-time such a constraint must be added. `A` is the class that defines the implementation that might be called on the object labeled l_o, and l_1 and l_2 are the labels of the actual and formal parameter, respectively. The number of resulting constraints is immense – namely at a given call site $\mathtt{o}^{l_o}.\mathtt{m}(\mathtt{a}^{l_1})$ we need for every callable method $\mathtt{m'(p}^{l_2}\mathtt{)}$ defined in class `A` conditional constraints $\mathtt{A} \in \widehat{\chi}(l_o) \Rightarrow \widehat{\chi}(l_1) \subseteq \widehat{\chi}(l_2)$ for every argument. These constraints check whether the class defining or inheriting $\mathtt{m'}$ is stored in $\widehat{\chi}(l_o)$. If it is, the set computed for, e. g., l_1 is unified with that stored in the formal parameter, l_2.

In the next chapter we will see that another property of conditional constraints is undesirable – they connect the constraints of different methods, namely the caller and the callee.

Having labeled all *interesting* program points, the constraints will model the flow of data through the program. The generation is performed by three functions. One is $C_\star$ for handling programs, method declarations and statements. The other two ($C^R_\star$ and $C^L_\star$) handle using and defining occurrences of expressions. The definitions are given in Table 4.7 for $C_\star$ and in Table 4.8 for $C^R_\star$ and $C^L_\star$. The types of these functions are

$$
\begin{aligned}
C_\star &: \textbf{Term} \to \mathcal{P}(\textbf{Constraints}) \\
C^R_\star &: \textbf{Rhs} \times \mathbb{N} \to \mathcal{P}(\textbf{Constraints}) \\
C^L_\star &: \textbf{Var} \times (\mathbb{N} \cup \{\epsilon\}) \to \mathcal{P}(\textbf{Constraints})
\end{aligned}
$$

The rules for constraint generation immediately follow from the definition of the syntax directed CFA in Tables 4.5 and 4.6. The additions are those constructs that model the *dynamic* behavior, that is method calling and access to class members. They are modeled by three functors, namely `LOAD`, `STORE`, and `CALL`. As explained above, the arguments to these functors are the labels and the type information that they need to establish flow to and from instance or class members and between caller and callee. While these additions are relatively small, their impact is immense as we will see in Chapters 5 and 8.

It is important to stress that the result of $C_\star$, $C^R_\star$, and $C^L_\star$ are sets of *syntactic* constructs. Namely, the $\widehat{\chi}$ and $\widehat{H}$ are translated into rules how to construct a constraint system. To fill this syntax with live, we would need to translate the C and `LOAD`, `STORE`, and `CALL` back into $\widehat{\chi}$ and $\widehat{H}$, c. f. [NNH1999]. This translation is done by means of the following rules.

$$
\begin{aligned}
(\widehat{\chi}, \widehat{H})[\![C(l)]\!] &= \widehat{\chi}(l) \\
(\widehat{\chi}, \widehat{H})[\![\{\mathtt{C}\}]\!] &= \{\mathtt{C}\} \\
(\widehat{\chi}, \widehat{H})[\![\mathtt{LOAD}\ (\mathtt{m}, \mathtt{C}, l_o, l_r)]\!] &= \forall \mathtt{C'} \in \widehat{\chi}(l_o) : \widehat{H}(\mathtt{C'}, \mathtt{m}, \mathtt{C}) \subseteq \widehat{\chi}(l_r) \\
(\widehat{\chi}, \widehat{H})[\![\mathtt{STORE}\ (\mathtt{m}, \mathtt{C}, l_o, l_r)]\!] &= \forall \mathtt{C'} \in \widehat{\chi}(l_o) : \widehat{\chi}(l_r) \subseteq \widehat{H}(\mathtt{C'}, \mathtt{m}, \mathtt{C})
\end{aligned}
$$

$$(\widehat{\chi},\widehat{H})[\![\texttt{CALL}\ (l_o,l_r,[l_1,\ldots,l_n],\texttt{C},AT)]\!] = \forall \texttt{C}' \in \widehat{\chi}(l_o):\ \text{if } \mathit{MostSpecial}(\texttt{m},\texttt{C}',\texttt{AT}) = \{(\texttt{C}'',MT',\texttt{T}_r\texttt{m}(\texttt{T}_1\ \texttt{x}_1^{lp_1},\cdots,\texttt{T}_n\ \texttt{x}_n^{lp_n})\{\mathit{Stmts}\}^{l_r})\}$$
$$\text{then } \forall 1 \le i \le n : \widehat{\chi}(l_i) \subseteq \widehat{\chi}(lp_i) \wedge \widehat{\chi}(l_r) \subseteq \widehat{\chi}(l)$$

Finally, we need an additional satisfaction relation $(\widehat{\chi},\widehat{H}) \models_C (lhs \subseteq rhs)$ on constraints:

$$\begin{array}{rcl}(\widehat{\chi},\widehat{H}) \models_C (\mathit{lhs} \subseteq \mathit{rhs}) & \Leftrightarrow & (\widehat{\chi},\widehat{H})[\![\mathit{lhs}]\!] \subseteq (\widehat{\chi},\widehat{H})[\![\mathit{rhs}]\!] \\ (\widehat{\chi},\widehat{H}) \models_C \mathit{functor} & \Leftrightarrow & (\widehat{\chi},\widehat{H})[\![\mathit{functor}]\!]\end{array}$$

In order to deal with the set of constraints generated for a program, we lift $\models_C$ to work on sets $M \subseteq \mathcal{P}(\textbf{Constraints})$:

$$(\widehat{\chi},\widehat{H}) \models_C M \text{ iff } \forall c \in M : (\widehat{\chi},\widehat{H}) \models_C c$$

Equivalently to [NNH1999] we then get the following result.

Theorem 4.5 *If* $(\widehat{\chi},\widehat{H}) \sqsubseteq (\widehat{\chi}_\star,\widehat{H}_\star)$ *then* $(\widehat{\chi},\widehat{H}) \models_S e_\star \Leftrightarrow (\widehat{\chi},\widehat{H}) \models_C C_\star[e_\star]$

Proof The proof is by structural induction on the term e to show that $(\widehat{\chi},\widehat{H}) \models_S e$ iff $(\widehat{\chi},\widehat{H}) \models_C C_\star[e]$. ∎

To conclude this chapter we return to Example 3.6. We will for each procedure give the constraints and the value of function *labels* that are generated by the control flow analysis.

Example 4.3 *(Constraints for the Example Program)*
We start with the constructor `Object.Object` *as this is the easiest case – only labels needs to be defined, as the method body is empty.*

$$\mathit{labels}(\texttt{Object.Object: void} \rightarrow \texttt{cstr}) = (\epsilon,[14])$$

For constructor `Election.Election` *we get the constraints that let the arguments abstractions flow to the* `CALL` *functor and the object of the access to the field* `count`.

$$\mathit{labels}(\texttt{Election.Election: void} \rightarrow \texttt{cstr}) = (\epsilon,[1])$$
$$\widehat{\chi}(1) \subseteq \widehat{\chi}(2) \qquad \widehat{\chi}(1) \subseteq \widehat{\chi}(3) \qquad \texttt{CALL}(\texttt{Object},2,\epsilon,[3],\texttt{Election},[])$$
$$\widehat{\chi}(1) \subseteq \widehat{\chi}(4)$$

The next procedure is `Election.tick`*. The constraints only resemble the flow of the argument of the method to the objects on that the access to the member is based.*

$$\mathit{labels}(\texttt{Election.tick: void} \rightarrow \texttt{void}) = (\epsilon,[5])$$
$$\widehat{\chi}(5) \subseteq \widehat{\chi}(6) \qquad \widehat{\chi}(5) \subseteq \widehat{\chi}(7)$$

The constraints for the two methods defined in class `Election2` *almost look the same as the two as those just seen:*

$$\mathit{labels}(\texttt{Election2.Election2: void} \rightarrow \texttt{cstr}) = (\epsilon, [8])$$
$$\widehat{\chi}(8) \subseteq \widehat{\chi}(9) \qquad \widehat{\chi}(8) \subseteq \widehat{\chi}(10) \qquad \texttt{CALL}(\texttt{Election}, 9, \epsilon, [10], \texttt{Election2}, [])$$
$$\mathit{labels}(\texttt{Election2.tick: void} \rightarrow \texttt{void}) = (\epsilon, [11])$$
$$\widehat{\chi}(11) \subseteq \widehat{\chi}(12) \quad \widehat{\chi}(11) \subseteq \widehat{\chi}(13)$$

Finally, we give the constraints for the method `App.main`*. Here, we model the creation of two objects, the flow to the constructor calls and finally to the calls of method* `tick`*.*

$$\mathit{labels}(\texttt{App.main: void} \rightarrow \texttt{void}) = (\epsilon, [])$$
$$\{\texttt{Election}\} \subseteq \widehat{\chi}(16) \qquad \widehat{\chi}(16) \subseteq \widehat{\chi}(15)$$
$$\widehat{\chi}(15) \subseteq \widehat{\chi}(17) \quad \widehat{\chi}(15) \subseteq \widehat{\chi}(18) \quad \texttt{CALL}(\texttt{Election}, 17, \epsilon, [18], \texttt{Election}, [])$$
$$\{\texttt{Election2}\} \subseteq \widehat{\chi}(20) \qquad \widehat{\chi}(20) \subseteq \widehat{\chi}(19)$$
$$\widehat{\chi}(19) \subseteq \widehat{\chi}(21) \quad \widehat{\chi}(19) \subseteq \widehat{\chi}(22) \quad \texttt{CALL}(\texttt{Election2}, 21, \epsilon, [22], \texttt{Election}, [])$$
$$\widehat{\chi}(19) \subseteq \widehat{\chi}(23) \quad \widehat{\chi}(19) \subseteq \widehat{\chi}(24) \quad \texttt{CALL}(\texttt{tick}, 23, \epsilon, [24], \texttt{Election}, [])$$
$$\widehat{\chi}(15) \subseteq \widehat{\chi}(25) \quad \widehat{\chi}(15) \subseteq \widehat{\chi}(26) \quad \texttt{CALL}(\texttt{tick}, 25, \epsilon, [26], \texttt{Election}, [])$$

We want to stress once more that none of the constraints for one procedure depends on those of another one. By detangling the constraints it is possible to compute them independently and store them procedure-wise. Thus they can be recombined as they are needed for analyzing a program.

Chapter 5

The Demand Driven Solver

No tactic ever so good is as successful as a dumb goal.

Günther Netzer, Soccer World Champion

This chapter will present the algorithm to compute a solution to the constraint system. Up to now we have seen how the constraints are generated from a program. They will now be transformed to a graph representation that lends itself to a better handling by the solver.

As already pointed out, when generating constraints for a whole program there exist no edges that model data flow between two methods. Those constraints will only be generated while the solution is computed. As a result, we may compute the constraint set and the graph representing it separately for every method. This allows to keep the constraint graph small since only those procedures are present, that already have been called. On demand, that is whenever a not yet inserted procedure shall be called, the nodes representing the constraints for the procedure are added to the graph.

We start by describing how the graph for a method is constructed. The resulting graph typically will be suboptimal, that is it may contain superfluous nodes and edges. So the next step is to optimize the graph. The result can potentially be cyclic. As we deal with set constraints and unification, the nodes will then be replaced by strongly connected components. This will give us an acyclic graph. The chapter concludes with the solver algorithm and the continuation of the example.

5.1 Generating Constraint Graphs

In order to efficiently solve the constraints generated by the techniques from the last chapter, we first transform them. A common representation for efficient solutions is a directed graph

where, in our case, the nodes represent the $C(l)$ and functor occurrences. The edges represent the dependencies between labeled program points. Additionally, the graph has a function D that maps nodes to the set of classes that has been computed for the node.

Definition 5.1 *(Constraint Graph, Functions type, D)*
Assume p a Bahasa method and $M := C_\star(p)$ the set of constraints generated for p. The graph $G_M = (N_M, E_M)$ is the **constraint graph** *with $E_M \subseteq N_M \times N_M$. Function type : $N_M \rightarrow \mathbb{N} \cup$ **Functors** returns either the label or the functor represented by a node. Function $D : N_M \rightarrow \mathcal{P}($**Classes**$)$ stores the abstract values associated with a node.*

G, D and type are initialized from M as follows:

- $\forall(\{\mathtt{C}\} \subseteq \widehat{\chi}(l)) \in M$:

  $n_l \in M \wedge D(n_l) = \{\mathtt{C}\}$
- $\forall(\widehat{\chi}(l_1) \subseteq \widehat{\chi}(l_2)) \in M$:

  $n_{l_1}, n_{l_2} \in N_M \wedge (n_{l_1}, n_{l_2}) \in E_M \wedge \mathit{type}(n_{l_1}) = l_1 \wedge \mathit{type}(n_{l_2}) = l_2$
- $\forall(\mathtt{LOAD}(\mathtt{f}, \mathtt{C}, l_1, l_2)) \in M$:

  $n_L, n_{l_o}, n_l \in N_M \wedge (n_{l_o}, n_L) \in E_M \wedge$

  $\mathit{type}(n_L) = \mathtt{LOAD}(\mathtt{f}, \mathtt{C}, l_1, l_2) \wedge \mathit{type}(n_{l_o}) = l_o \wedge \mathit{type}(n_l) = l$
- $\forall(\mathtt{STORE}(\mathtt{f}, \mathtt{C}, l_1, l_2)) \in M$:

  $n_S, n_{l_o}, n_l \in N_M \wedge (n_{l_o}, n_S) \in E_M \wedge$

  $\mathit{type}(n_S) = \mathtt{STORE}(\mathtt{f}, \mathtt{C}, l_1, l_2) \wedge \mathit{type}(n_{l_o}) = l_o \wedge \mathit{type}(n_l) = l$
- $\forall(\mathtt{CALL}(\mathtt{m}, l_o, l_r, [l_{p_1} \cdots l_{p_i}], \mathtt{C}, AT)) \in M$:

  $n_C, n_{l_o}, n_{l_r}, n_{l_1}, \cdots, n_{l_{p_i}} \in N_M \wedge (n_{l_o}, n_C) \in E_M \wedge$
  $\mathit{type}(n_{l_o}) = l_o \wedge \mathit{type}(n_{l_r}) = l_r \wedge \mathit{type}(n_{l_1}) = l_1 \wedge \cdots \wedge \mathit{type}(n_{l_{p_i}}) = l_{p_i} \wedge$
  $\mathit{type}(n_C) = \mathtt{CALL}(\mathtt{m}, l_o, l_r, [l_{p_1} \cdots l_{p_i}], \mathtt{C}, AT)$

We will need to classify nodes in a constraint graph. *Special* nodes are functor nodes or nodes that represent a formal parameter, the label of a method body or the source (target) label of a STORE (LOAD) functor.

Definition 5.2 *(Special Nodes)*
Let p be a Bahasa program containing procedures $P_1, \cdots, P_l$, $G = (N, E)$ be a constraint graph. node $\in N$ is called **special***, if it fulfills one of the following conditions*

- *type*(*node*) $\in \{$*CALL*, *LOAD*, *STORE*$\}$

or with $l_{node} =$ type(node)

- $\exists 1 \leq i \leq l : labels(signature(P_i)) = (l_r, [l_1, \cdots, l_n]) \wedge l_{node} \in \{l_r, l_1, \cdots, l_n\}$
- $\exists n \in N, \mathit{type}(n) = \texttt{STORE}(\texttt{f}, \texttt{C}, l_o, l_{node}) \vee \mathit{type}(n) = \texttt{LOAD}(\texttt{f}, \texttt{C}, l_o, l_{node})$

We define a function special : $N \rightarrow \{\texttt{true}, \texttt{false}\}$ *with* $special(n) = true \Leftrightarrow n$ *is special.*

In addition, there is the function $\widehat{H}$: **Classes** $\times$ **Members** $\times$ **Classes** $\rightarrow \mathcal{P}$(**Classes**) from the last chapter that stores information for class and instance members.

Note how the edges represent dependencies between labeled points in the program. E. g., the interpretation of an edge between the label l_o of an object and a call node n_C is, that whenever the set of classes $D(n_{l_o})$ associated with the object changes, the solver will need to revisit the node n_C. This will be handled in more detail in Section 5.4.

Example 5.1 *(Constraint Graph for the Example Program)*
Figure 5.1 shows the constraint graphs generated for the methods of the example program. The upper half of each node shows type$(\cdot)$*, the lower half* $D(\cdot)$*. The gray boxes group the nodes belonging to the same procedure.*

5.2 Optimizing Constraint Graphs

As can be seen in Figure 5.1, the generated graphs tend to contain superfluous nodes and edges. One of the reasons for this are the requirements we have imposed on the structure of `Bahasa` programs in Chapter 3, namely that all values used as, e. g., arguments to method calls are first assigned to local variables. In order to minimize the graph size we run a graph compacting algorithm before starting the solver. This step is performed by the algorithm *optimize* given in Table 5.1. The algorithm performs three phases.

1.) First, it will delete those sequences of nodes that just propagate a value over a chain of edges through the constraint graph. However, it will leave *special* nodes untouched. The phase starts by collecting those nodes that have only one predecessor. The node (and its in- and outgoing edges) are then deleted and the predecessor of the node is connected to the successors. This approach can be compared to projection merging from [SFA2000]. Whenever the successor of a deleted node n_l is a functor node, all references of n_s to the label l are adjusted.

2.) The second phase starts with those nodes, that are not special and do not have successors. That is, these are those nodes that, e. g., represent the label of an object that is used for accessing class members that do not have class type. This can be seen in the example constraint graph in Figure 5.1. The nodes with labels 4, 6, 7, 12 and 13 represent those objects whose `count` field is accessed. As this field has type `int`, it will not influence the construction of the interprocedural control flow graph. Thus, these nodes can be deleted, a task performed by the second phase.

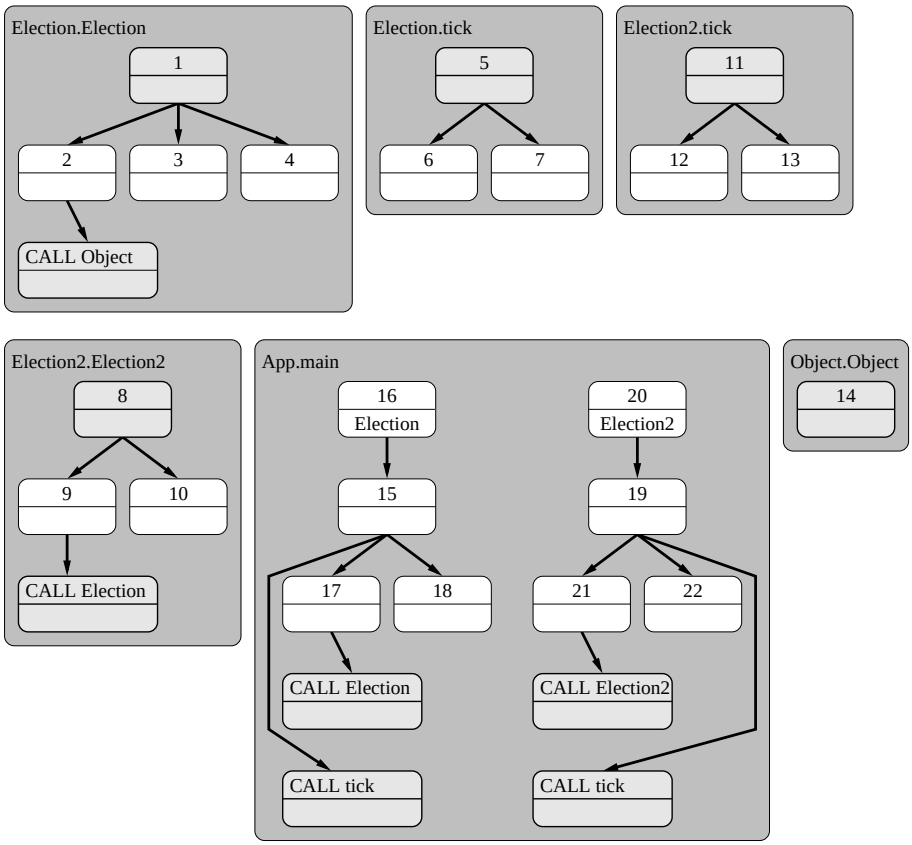


Figure 5.1: Constraint graph for the example program

Class	bytecode size	Nodes	Edges	opt. Nodes	opt. Edges
`java.lang.Object`	1514	123	110	45	15
`java.lang.System`	3436	1221	1075	424	163
`java.lang.Integer`	7910	735	673	252	97

Figure 5.2: Node and edge counts for Java libraries

3.) Finally, all isolated nodes that are not special are deleted.

Figure 5.2 shows some examples for graph sizes generated for Java library classes before and after optimization. The total size of the graph for *HelloWorld.java* is 25272 nodes and 22468 edges in the unoptimized graph and 10106 nodes and 4043 edges after applying `optimize`.

```
procedure optimize_graph(G = (N, E))
    worklist = [n ∈ N ∧ #predecessors(n) = 1 ∧ ¬special(n)];
    while (worklist ≠ []) {
        n = head(worklist); worklist = tail(worklist);
        succs = E|{n}×N;
        l = label(n);
        delete_node(n);
        ∀n_s ∈ succs{
            if (#predecessors(n_s) = 1 ∧ ¬special(n_s))
                worklist = append(worklist, n_s);
        }
    }
    worklist = [n ∈ N ∧ #successors(n) = 0 ∧ ¬special(n)];
    while (worklist ≠ []) {
        n = head(worklist); worklist = tail(worklist);
        n_p = predecessor(n);
        delete_node(n);
        if (#successors(n_p) = 0 ∧ ¬special(n_p))
            worklist = append(worklist, n_p);
    }
    forall n ∈ N
        if (¬special(n) ∧ E|{n}×N ∪ E|N×{n} = ∅)
            delete_node(n, G);
end

procedure delete_node(n, G = (N, E))
    N = N\{n};
    n_p = n_1, where {n_1} = {n_2|e ∈ E|N×{n}};
    succs = {n_s|(n, n_s) ∈ E};
    E = E ∪ {(n_p, n_s)|n_s ∈ succs)
    forall n_s ∈ succs {
        if type(n_s) ∈ Functors ∧ label(n) ∈ n_s
            replace label(n) with label(n_p)
    }
end
```

Table 5.1: Optimizing constraint graphs

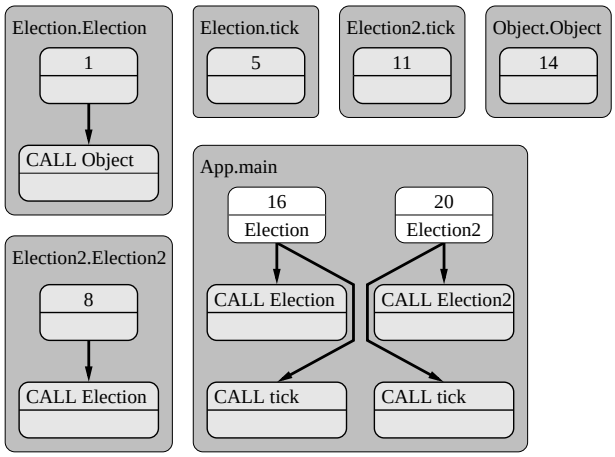


Figure 5.3: Optimized constraint graph for the example program

Example 5.2 *(Optimized Constraint Graph)*
Figure 5.3 shows the optimized constraint graph of the example program. Note how all the intermediate nodes have vanished.

It should be noted that it is not typical that almost all non special nodes disappear from the graph. Every non special node that represents a label with more than one defining label, that is more than one incoming edge, will still be present.

5.3 Transformation to Directed Acyclic Graphs

The constraint graph we have constructed and especially the graphs that arise during the solving phase might contain cyclic dependencies between nodes. As the nodes and edges resemble sets and inclusion constraints between those sets, we know that all nodes that lay in a cycle will end up with the same set of classes, namely the union of the sets of those nodes. This is why we compute in a next step the graph containing the strongly connected components of the constraint graph. This is done by applying one of the standard construction algorithms for strongly connected components, e. g. [CLR1992].

Later on, during solving the constraint system, edges may be added, eventually resulting in a cyclic graph. However, the algorithm will make sure that newly created cycles are detected and collapsed to a new strongly connected component. Thus the graph remains acyclic.

Definition 5.3 *(Strongly Connected Components Graph, Functions scc, $nodes$, D_{scc}, $type_{scc}$)*

Let $G_{\mathcal{C}} = (N_{\mathcal{C}}, E_{\mathcal{C}})$ be a constraint graph.
$\widehat{G_{\mathcal{C}}} = (\widehat{N_{\mathcal{C}}}, \widehat{E_{\mathcal{C}}})$ is the graph consisting of the strongly connected components of $G_{\mathcal{C}}$. Functions $scc : N_{\mathcal{C}} \to \widehat{N_{\mathcal{C}}}$, $nodes : \widehat{N_{\mathcal{C}}} \to \mathcal{P}(N_{\mathcal{C}})$ return the strongly connected component for a constraint node respectively the constraint nodes that belong to such a component.
The function $D_{scc} : (N_{\mathcal{C}} \cup \widehat{N_{\mathcal{C}}}) \to \mathcal{P}(\textbf{\textit{Classes}})$ is defined by

$$D(n) := \begin{cases} \bigcup_{n' \in nodes(n)} D(n') & \text{if } n \in \widehat{N_{\mathcal{C}}} \\ \textit{type}(n') & \text{otherwise} \end{cases}$$

The function $\textit{type}_{scc} : (N_{\mathcal{C}} \cup \widehat{N_{\mathcal{C}}}) \to (\mathbb{N} \cup \textbf{\textit{Functors}} \cup \{normal\})$ is defined by

$$\textit{type}_{scc}(n) := \begin{cases} \textit{type}(n) & \text{if } n \in N_{\mathcal{C}} \\ \textit{type}(n') & \text{if } n \in \widehat{N_{\mathcal{C}}} \wedge nodes(n) = \{n'\} \wedge \textit{type}(n') \in \textbf{\textit{Functors}} \\ \textit{normal} & \text{else} \end{cases}$$

We will see later that the special nodes `CALL`, `LOAD`, and `STORE` share a special property: they will always be the single node in their strongly connected component as they may never have outgoing edges. This allows us to speak of *normal*, *call*, *load*, and *store* components, depending on the nodes contained within, just as we have done for nodes in the constraint graph. Let $\widehat{G_{\mathcal{C}}}$ be the graph containing the strongly connected components of $G_{\mathcal{C}}$. Obviously, $\widehat{G_{\mathcal{C}}}$ is acyclic. Using one of the standard algorithms like that in [NNH1999], this property allows us to compute the topological order on its nodes. This defines a function $top : \widehat{N_{\mathcal{C}}} \to \mathbb{N}$ that will guide the solver in selecting nodes to work on. Like the acyclic property, the topological ordering is maintained when edges are inserted.

5.4 Solving the Constraint System

Having constructed the optimized constraint graph, we apply the solving algorithm. It is worklist based and will propagate already computed values of D_{scc} along the edges. As described below whenever a functor node is hit the solver may add edges to the constraint graph. We will start by describing some helper procedures used by the solver. These are

- $addtoworklist$ that is in charge of maintaining a clever ordering on the nodes in the worklist (Table 5.2)
- $addedge$ that adds edges, checks for cycles in the graph and readjusts top (Table 5.3)
- $membernode$ and $lookup$ that handle functor nodes (Table 5.5)

The algorithm itself is given in Table 5.6

```
procedure addtoworklist(n, worklist)
    if (n ∈ worklist)
        return;
    t = worklist.initialnodes;
    if (¬is_initial(n)) {
        switch (type(n)) {
            case STORE :
                t = worklist.storenodes; break;
            case LOAD :
                t = worklist.loadnodes; break;
            case CALL :
                t = worklist.callnodes; break;
            default:
                t = worklist.normalnodes; break;
        }
        while (type(t) = type(n) ∧ top(t) < top(n) ∧ t not last node)
            t = succ(t);
    }
    insert n after t;
end;
```

Table 5.2: Adding components to the worklist

5.4.1 Maintaining the Worklist

The worklist is a list that is divided in five parts with a link to the beginning of each part. Nodes are inserted depending on their type to one of these. The heuristic is that the nodes coming earlier in the worklist will have a higher impact on the graph. The order is

1. initial nodes
2. `STORE` nodes
3. `LOAD` nodes
4. `CALL` nodes
5. normal nodes

Initial nodes are the root nodes of the constraint graph. Those nodes n have a non empty set $D(n)$, that is, they represent object creations. Functor nodes may cause new edges to be added to the graph. If a `LOAD` functor is in the worklist, it should only be evaluated *after* visiting all `STORE` functors. Finally, `CALL` functors may result in new edges to normal nodes.

In addition, each of the worklist parts is ordered in increasing topological order. Thus any component n that might influence the value of another component n' is visited first. The procedure $addtoworklist$ checks the type of the component to add and inserts it in the right place. The pseudo-code is given in Table 5.2.

5.4.2 Adding Edges

Tables 5.3 and 5.4 show the procedures that handle the addition of edges to the graph. This process can be split in three phases. First, the edge is added. In a second phase, it must be checked whether the new edge introduces a cycle. If so, the components that are connected must be collapsed to a new component, resulting in an acyclic graph. This step is guided by the topological order on strongly connected components. If $top(\mathit{to}) < top(\mathit{from})$ then the added edge may be a back edge and may introduce a cycle. The nodes that are in the new component (if any) are collected and collapsed. The collection starts at *from* and checks whether *to* can be reached. This process also is guided by the topological order. Namely, an edge (n_1, n_2) may only be in a new component if $top(n_2) < top(to)$.

Finally, the third phase checks whether the topological order has been invalidated by the added edge or the restructuring. If so, the order must be recomputed for the subgraph S containing the edge. We first collect the numbers occurring as values of $top(n)$ for nodes in S and order them. Then the subgraph is traversed in topological order and the sorted numbers are assigned to the nodes, resulting in a valid topological ordering.

As can be seen, adding edges is handled quite efficiently. By keeping the topological order up to date, checking for cycles is kept simple.

Furthermore, the procedure takes care of adding the target of the new edge to the worklist. This is necessary, as by adding an edge a new possible flow of data may occur. Thus for the target we must recompute the value of $D(\cdot)$.

5.4.3 Handling Member Nodes and Functions

The solver will need to model the abstract run-time behavior introduced in Chapter 4. Each class or instance member is represented as a node in the graph. We need a function that returns the strongly connected component in which this node is contained. Additionally, we need a function to model the lookup mechanism for dynamic dispatching. Both these functions are given in Table 5.5.

Function $membernode$ gets as arguments

procedure $addedge(\mathit{from}, to, G = (N, E))$
    if ($\mathit{from} = to \vee (\mathit{from}, to) \in E$)
        return;
    $E = E \cup \{(\mathit{from}, to)\}$;
    $addtoworklist(to)$;
    if ($top(\mathit{from}) < top(to)$) return;
    forall $n \in N$ { $reaches(n) =$ `false`; $visited(n) =$ `false`; }
    $S = collectsccs(\mathit{from}, to, G)$;
    if ($S \neq \emptyset$) {
        $N = N \cup \{n_{scc}\}$;
        $D_{scc}(n_{scc}) = \bigcup_{n \in S} D_{scc}(n)$;
        $nodes(n_{scc}) = \bigcup_{n \in S} nodes(n)$; $\forall n \in nodes(n_{scc}) : scc(n) = n_{scc}$;
        $preds = \{n_p | (n_p, n) \in E|_{(N \setminus S) \times N}\}$;
        $succs = \{n_s | (n, n_s) \in E|_{N \times (N \setminus S)}\}$;
        $E = E|_{(N \setminus S) \times (N \setminus S)} \cup \{(n_p, n_{scc}) | n_p \in preds\} \cup \{(n_{scc}, n_s) | n_s \in succs\}$;
        $N = N \setminus S$;
        $to = n_{scc}$;
    }
    $adjusttop(to, G)$;
end;

procedure $collectsccs(start, target)$
    if ($reaches(start)$) return $\{start\}$;
    if ($visited(start)$) return $\emptyset$;
    $visited(start) =$ `true`;
    $M = \emptyset$;
    forall $n \in \{n_s | (start, n_s) \in E \wedge top(n_s) \leq top(target)\}$
        $M = M \cup collectsccs(n, target)$;
    if ($M \neq \emptyset \vee start = target$) {
        $M = M \cup \{start\}$;
        $reaches(start) =$ `true`;
    }
    return M;
end;

Table 5.3: Adding edges to the graph

```
procedure adjusttop(n, G = (N, E))
    forall n ∈ N visited(n) = false;
    (H, M) = gettreefragment(n, G);
    T = {i|s ∈ F ∧ ∃m ∈ nodes(s) : top(m) = i};
    forall n ∈ N visited(n) = false;
    forall h ∈ H
        T = renumber(h, T);
end;

procedure gettreefragment(n, G = (N, E))
    if (visited(n))
        return (∅, ∅);
    visited(n) = true;
    if (¬∃(n_p, n) ∈ E)
        H = {n};
    M = {n};
    forall p ∈ {n_p|(n_p, n) ∈ E}
        (H, M) = (H, M) ∪ gettreefragment(n_p);
    forall s ∈ {n_s|(n, n_s) ∈ E}
        (H, M) = (H, M) ∪ gettreefragment(n_s);
    return (H, M);
end;

procedure renumber(n, T)
    if (visited(n)) return;
    forall p ∈ {n_p|(n_p, n) ∈ E}
        if (¬visited(p))
            return;
    visited(n) = true;
    top(n) = min(T);
    T = T\{top(n)};
    forall s ∈ {n_s|(n, n_s) ∈ E}
        T = renumber(s, T);
    return T;
end;
```

Table 5.4: Recomputing the topological order

procedure $membernode(member, T_{run}, T_{static}, G = (N, E))$
    if ($\exists s \in N : n_{T_{run},T_{static},member} \in nodes(s)$)
        return s;
    $N = N \cup \{s_{new}\}$;
    $scc(n_{T_{run},T_{static},member}) = s_{new}$;
    return s_{new};
end;

procedure $lookup(method, D, argtype)$
    $targets = \bigcup_{\forall c \in D}$ *MostSpecial*$(method, c, argtype)$;
    forall $(C, methtype, body) \in targets$)
        $signatures = signatures \cup \{C.method : methtype\}$;
    return $signatures$;
end;

Table 5.5: Handling functor nodes

- the name `m` of the member
- a possible run-time type `C` for the object
- the static type `T` where the member has been declared
- the constraint graph

It returns the strongly connected component that contains the node n that represents the correct member. If this node does not yet exist, it is added.

Function $lookup$ gets as arguments

- the name `m` of the method to be called
- the possible run-time types `D` for the object
- the argument type *AT* of the method

For every class `C` in `D` the *MostSpecial*(`m`, `C`, *AT*) is computed and the set of signatures is returned.

5.4.4 The Solver

Table 5.6 shows the actual solver algorithm. It is worklist based and iterates over the strongly connected components graph.

The worklist is initialized with all the root components of entry points of the program being analyzed. Since the graph is still in its *pure* form, that is there have not yet been added any edges, each of this components represents exactly one node. By inspecting the rules for node generation in Section 5.1 one can see that these nodes are exactly those where objects are created. The boolean flag $initial$ ensures that these nodes are at least evaluated once.

In the iteration phase, the first node s from the worklist is chosen. Then the union of all the incoming abstract values is computed. If it differs from the original value $D(s)$ or $initial(s)$ was `true`, all successors of s are added to the worklist.

Finally, the type of component s is investigated, and depending on it different actions may occur. If the component is *normal*, i. e. only represents non-functor nodes, the iteration may continue. If the component contains a `LOAD` or a `STORE` node, all classes that have been computed for label l_o are inspected. Note that these classes exactly represent the possible run-time types of the object that is used for the member access. The node representing the member is identified by means of $membernode$ and an edge is added to let the values flow to or from this node n_m.

If component s represents a `CALL` node, we lookup the signatures of all possible targets p by means of function $lookup$. Function $addproc$ checks whether a callable procedure has already been inserted into the graph. If not, the constraints are computed, transformed to a graph and added to the graph. Then, edges are inserted into the graph in order to connect the nodes representing the call site with the nodes representing the target procedure. Finally, p is added to the set of called methods.

5.5 Applying the Results

Having solved the constraint graph we are now going to optimize the program using the solution. This requires two steps. First, we write back the results from strongly connected components to the nodes represented by them. Then, we annotate every call with the set of methods that are actually callable. This information can be used by the compiler backend to avoid generation of dynamically dispatched calls.

This optimization is useful for several reasons. As pointed out already, the call graph will be more precise and will enable faster and more precise data flow analyses. Additionally, we exactly know the target procedure for more call sites. This allows further optimizations like inlining. Last but not least, pipeline and cache analyses like [FHL+2001, FW1999] benefit for two reasons. On the one side, to perform the dynamic dispatch an array access is necessary that might dilute the cache. On the other side, the pipeline must be stalled, since the target of the call is only known after the array contents has arrived.

procedure $solve(\widehat{G_C})$
  let $\widehat{G_C} = (\widehat{N_C}, \widehat{E_C})$ be an empty constraint graph, $called = \emptyset$;
  forall entry procedures p { $addproc(\widehat{G_C}, p)$; $called = called \cup \{p\}$; }
  forall $s \in \widehat{N_C}$ with $\#predecessors(s) = 0 \wedge D(s) \neq \emptyset \wedge proc(s) \in called$
    $addtoworklist(s)$; $initial(s) =$ `true`;
  while $(worklist \neq [])$ {
    $s = head(worklist)$;$worklist = tail(worklist)$;
    $D_s = \bigcup\{D(p) | (p, s) \in \widehat{E_C}\}$
    if $(D_s = D(s) \wedge initial(s) =$ `false`$)$ continue;
    $D(s) = D_s$; $initial(s) =$ `false`;
    forall $(s, n) \in E$ $addtoworklist(n)$;
    switch *type*(s) {
      normal: break;
      `LOAD` (`m`,`C`,l_o,l):
        forall `T` $\in D(scc(n_{l_o}))$ {
          $addedge(membernode(\mathtt{m}, \mathtt{T}, \mathtt{C}, \widehat{G_C}), scc(n_l), \widehat{G_C})$;
        break;
      `STORE` (`m`,`C`,l_o,l):
        forall `T` $\in D(scc(n_{l_o}))$
          $addedge(scc(n_l), membernode(\mathtt{m}, \mathtt{T}, \mathtt{C}, \widehat{G_C}), \widehat{G_C})$;
        break;
      `CALL` (`m`,l_o,l_t,[l_1,$\cdots$,l_n],`C`,*AT*):
        $targets = lookup(\mathtt{m}, D(scc(n_{l_o})), AT)$;
        forall $p \in targets$ where $labels(p) = (l_r, [p_1, \cdots, p_n])$ {
          if $(p \notin called)$ { $addproc(\widehat{G_C}, p)$; $called = called \cup \{p\}$;
          for $1 \leq i \leq n$: $addedge(scc(n_{l_i}), scc(n_{p_i}), \widehat{G_C})$
          $addedge(scc(n_{l_r}), scc(n_{l_t}), \widehat{G_C})$;
          $called = called \cup \{p\}$;
          }
          break;
    }
  }
end;

Table 5.6: Solving constraint graphs

procedure $writeback(G = (N, E), \widehat{G_C} = (\widehat{N_C}, \widehat{E_C}), scc)$ {
  for all $n \in N$
    if (*type*$(n) \notin$ **Functors**)
      $\widehat{\chi}$(*type*$(n)) = D(scc(n))$;
  for all $\mathtt{m} \in$ **FNames**, $\mathtt{C}, \mathtt{C}' \in$ **Classes**
    if ($membernode(\mathtt{m}, \mathtt{C}, \mathtt{C}') \in \widehat{N_C}$)
      $\widehat{H}(\mathtt{C}, \mathtt{m}, \mathtt{C}') = D(membernode(\mathtt{m}, \mathtt{C}, \mathtt{C}'))$;
}

Table 5.7: Writing back analysis results

5.5.1 Writing back the Results

Having computed a solution for the constraint graph, we need to write back the results. Remember that the labels originally assigned to program points have been transformed to nodes that may have been combined to strongly connected components during the solving process.

We need to make sure that the computed sets are written back to the correct label. The procedure that performs this task is given in Table 5.7. It uses the function scc to look up the strongly connected component a node n_l has been joined with. The set computed for the component is written back to $\widehat{\chi}(l)$. Equivalently, the information computed for member nodes is stored in $\widehat{H}$.

5.5.2 Optimizing the Program

Having reconstructed $\widehat{\chi}$ and $\widehat{H}$ from the constraint graph, we use this information to optimize the program. The optimization is based on the computed set $\widehat{\chi}(l_o)$, where l_o labels an object that is used to call a method. We start by computing the set of callable methods for $\widehat{\chi}(l_o)$.

Definition 5.4 *(Function Callable)*
Let $C \subseteq$ ***Classes***. *Function Callable* : (***MNames*** $\times$ ***Classes*** $\times$ ***Types****) $\rightarrow$ $\mathcal{P}$(***Classes*** $\times$ $\bigcup_{n\geq 0}$ ***Types***^{n} $\times$ ***Term***) *is defined by:*

$$\mathit{Callable}(\mathtt{m}, M, AT) = \bigcup_{\mathtt{C} \in M} \mathit{MostSpecial}(\mathtt{m}, \mathtt{C}, AT)$$

Given a call statement $s \equiv \mathtt{o}^{l_o}.\mathtt{[AT]m}(\ldots)$ and $M_s = \mathit{Callable}(\mathtt{m}, \widehat{\chi}(l_o), AT)$, there are several cases regarding the size of M_s to consider:

1. the set is empty, that is the call is never reached
2. it is a singleton set, that is exactly one method can be called
3. there exist several callable procedures

In the first case, the compiler might warn the user about an unreachable statement or a call on an uninitialized object.

In the second case, the compiler can generate a static call instead of the dynamic dispatch. As motivated in the introduction, this enables or improves further analyses and optimizations like inlining or cache and pipeline analysis.

The third case allows to apply some more heuristics, depending on the actual size of the set. There has some work been done on replacing the dynamic call with `if ... then ... else ...` constructs [AH1996, DGC1995]. The dispatch process is encoded into the generated code, with each resulting call site having exactly one target method. The conditions check the run-time type of the object. In the example program from Example 3.1, the call `candB`^{21}`.[]Election2(candB`^{22}`);` can be coded as:

```
if (candA instanceOf Election)
   ((Election)candA).[]tick(candA);
else
   ((Election2)candA).[]tick(candA);
```

In this case, we need to use the cast of `candA` to `Election` respectively `Election2` in order to enable the type checking phase of the compiler to determine the actual types that are ensured by the `if` clauses.

5.6 Analyzing Libraries

We have already stated several times that the created constraint graphs can be partitioned. This section will in detail present how this works.

Example 4.3 has given a practical example of the constraints generated for a program. The according constraint graphs are given in Figures 5.3 and 5.3. Here, the modularization is emphasized by grouping the nodes for the methods.

By inspecting the rules for constraint generation, one can prove the following theorem.

Theorem 5.1 *Let* p *be a* `Bahasa` *program with methods* $m_1, \ldots, m_n$ *and* $C_p \in \mathcal{P}(\textbf{\textit{Constraints}})$: $C_p = C_\star(\mathtt{p})$ *the set of constraints generated for* p.
C_p *can be divided into disjoint sets* $C_i \in \mathcal{P}(\textbf{\textit{Constraints}})$: $C_i = C_\star(\mathtt{m}_i)$ *for* $1 \leq i \leq n$ *and* $C_p = \dot{\bigcup}_{1 \leq i \leq n} C_i$.

Proof By inspecting the rules from Tables 4.7 and 4.8, the definition of rd and du as *intraprocedural* analyses one easily sees that no constraint graph for one procedure depends on that of another procedure. The only rules where some dependence could occur are those for member access and method call. However, these have been modeled as functor nodes that separate the calling procedure from possible target procedures. ∎

The steps for precompiling a method `m` are

- generate constraints C_m for `m`
- construct constraint graph G_m
- compute the optimized constraint graph G'_m
- compute the strongly connected graph G''_m
- save this graph according to the paths in the Java standard libraries

Example 5.3 *(Precomputed Graph for the Example Program)*
The precomputed graph for the constructor `Election2` *would be stored in a file* `Election/Election__.cst`.

The contents of the file is given in Figure 5.4. It states that the method for which the constraints file has been generated is named `Election2` *and that it has 2 nodes. The mapping computed for the method,* $labels($`Election2`$)$*, was* $-1, (1)$ *as the constructor does not return a value and the label assigned to the one argument is 1. Next follows a list of nodes in the graph; there are two of them. The first is the node with label 1, representing the argument of the method. The second node is the call to the constructor* `Election` *of the super class. Note how the arguments to the* `CALL` *functor are all present in the file. Furthermore, there is an edge from the first to the second node, as the call is based on the argument to method* `Election2`.

As there is no back edge, each of the nodes constitutes its own strongly connected component. This is stated by the next two lines. Finally, we need to state which edges exist in the graph of strongly connected components. There is one edge from the component containing the argument node to that containing the call node.

The construction phase of the constraint graph for the whole program is slightly modified. Originally, the rule $[\widehat{Program}]_C$ from Table 4.7 initiated the constraint construction for the whole program `p`, that is for all methods in all classes in `p`. Now, rule $[\widehat{MethodDefinition}]_C$ needs to be modified. It first checks, whether there exists a precompiled file for the method. If that file exists, it is checked whether the file is still up to date, e. g. by inspecting the file creation date or by computing a hash value on the method and storing it in the precompiled file. If the file still is valid, the constraint graph is constructed from the file and added to the graph computed for `p`. If not, the method is analyzed just as described in this chapter.

Chapter 8 will show the impact that this precompilation has on the total analysis time for rapid control flow analysis.

```
PROC Election2__ 2
LABELS -1,(1)
# nodes
N     1 1
CALL  2 Election__(1,-1,(1),java_lang_Election2,
                        void||java_lang_Election|)
# edges
E     1 2
# strongly connected components
SZK         1 1
SZK         2 2
# successor of strongly connected components
SSUCCS      1 2
SSUCCS      2
```

Figure 5.4: Precompiled information for method `Election2`

5.7 Complexity

Finally we want to inspect the overall complexity of our approach. To start, we first repeat the phases needed:

- label interesting expressions
- compute intraprocedural reaching-definitions and definition-use chains rd and du on labels
- generate constraints
- solve constraints

The labeling is done during parsing and type-checking the program. Its complexity is $O(n)$ as we need to inspect every expression exactly once. n is the size of the program. The complexity for reaching-definitions and definition-use chains is $O(n)$ for both analyses. For generating constraints, again we need one pass over the program source code, that is complexity is $O(n)$. Finally we need to solve the constraint graph. The number of constraints generated is $O(n)$ for the functor constraints and $O(n^2)$ for the inclusion constraints.

Following [NNH1999] the following theorem can be proven. It states that the solver will terminate and actually compute a least solution.

Theorem 5.2 *Let* p *be a* `Bahasa` *program,* $C_\star[p]$ *the set of constraints constructed for* p*,* $\widehat{G_\mathcal{C}} = (\widehat{N_\mathcal{C}}, \widehat{E_\mathcal{C}})$ *the graph of strongly connected components constructed for* p*. The algorithm from*

Table 5.6 terminates on $\widehat{G_C}$ *and the result* $(\widehat{\chi}, \widehat{H})$ *computed satisfies*

$$(\widehat{\chi}, \widehat{H}) = \bigsqcap\{(\widehat{\chi}', \widehat{H}') | (\widehat{\chi}', \widehat{H}') \models_C C_\star[p]\}$$

making it the least solution to $C_\star[p]$.

Furthermore, the run-time of the algorithm can be bounded. The initialization takes time $O(n)$, the graph construction takes time $O(n^2)$ due to the number of constraints. Optimizing the graph will take time $O(n)$, since we only visit such nodes with one predecessor or no successor. The writing back of solution after solving the graph will take time $O(n)$. Still left is the loop that handles the worklist. The graph may have at most $O(n^2)$ edges. The number of times they are visited is limited by the number of classes in the program, that is $O(n)$ again. Thus, the overall time needed by the algorithm is $O(n^3)$.

However, this is the worst case, if no strongly connected components are found in the graph. Furthermore, by far not all methods in the graph will be visited and the height of our abstract domain, the powerset of the defined classes, usually is rather small.

Chapter 6

Extending the Language with Exceptions

In this chapter another important feature of object-oriented languages is added to `Bahasa`.

Exceptions not only allow handling of erroneous situations, like trying to access members of a variable that contains `null` instead of a valid address of an object. They also can be used by the programmer in two ways. On the one side the compiler can be told about errors that possibly may occur in a method. Thus at compile-time can be checked that the program really handles such exceptions. On the other side the programmer also can define her own exceptions and use them as a replacement for e. g. `goto`.

We will first add exception-handling to the syntax of `Bahasa`. Additionally configurations, $\texttt{Bahasa}_r$ and the operational semantics from Chapter 3 will be extended. Finally, we extend the analysis to trace exceptions additionally to *normal* data.

Table 6.1 shows the extensions that are added to `Bahasa`. Actually this includes the possibility to state for a method that it may throw an exception as well as statements for throwing exceptions and catching them. Note that while executing a Java program there is always only one *active* exception that has been thrown and has not yet been caught. This is modelled in the semantics of `Bahasa`. Namely, there is a new field in the configuration. This field contains either ϵ or the name

MethodDefinition	:=	*Modifier ReturnType* MethodId ((*VarType* ParamId^{l_i})*) [throws (ClassName)$^+$] { *LocVariableDefinition** *Stmts* ; }^{l_r}
Stmt	:=	`throw VarId`^{l}
	|	`try` *Block* (`catch ClassName VarId`^{l} *Block*)* `finally` *Block*
	|	`try` *Block* (`catch ClassName VarId`^{l} *Block*)$^+$

Table 6.1: Exceptions in `Bahasa`

$$[\textit{throws 1}] \quad \frac{\left\langle \texttt{VarName}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \textit{val}, \sigma, H, \epsilon \right\rangle}{\left\langle \texttt{throw VarName}, \sigma, \texttt{H}, \epsilon \right\rangle \rightsquigarrow_{\texttt{P}} \left\langle \texttt{throw } \textit{val}, \sigma, \texttt{H}, \epsilon \right\rangle}$$

$$[\textit{throws 2}] \quad \frac{}{\left\langle \texttt{throw } \iota_i^E, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma, H, \iota_i^E \right\rangle}$$

Table 6.2: Extended Bahasa semantics for throwing exceptions

of the exception class that has been thrown by the program. The definition of a configuration is changed to be

Configuration ::= $\langle t, \sigma, H, E \rangle$, $t \in$ **Term** $\cup \{\epsilon\}$, $\sigma \in$ **Env**, $H \in$ **Heap**, $E \in \{\epsilon\} \cup$ **Classes**

The field E will be set by the semantics of the `throw` statement and will be read by the `catch` statement.

The Java language specification [GJS1996] contains many cases where the run-time system is supposed to throw an exception. These include *null pointer exceptions* when a variable does contain `null` instead of a valid reference and *index out-of-bounds* exceptions when an array is accessed with an invalid index. We assume that in Bahasa programs is checked for all these cases, just like a Java virtual machine would do. That is, a method send expression like

```
o.method(o);
```

would be translated into

```
if (o!=null) then
  o.method(o);
else {
  e=new NullPointerException();
  e.NullPointerException(e);
  throw e;
}
```

This externalisation of exceptions allows to handle the usually implicit exceptions like *null pointer* or *index out of bounds* similar to user defined exceptions.

Table 6.2 contains the rules needed to handle the cases where an exception may be thrown. Additionally there must be rules for catching and propagating exceptions. After an exception has been thrown, it must be propagated to either the catch clauses of a surrounding `try` statement or the end of the procedure. As can be seen in Table 6.3, we can easily identify in which situation the exception occurs – either, we are in a `try` block, then we check the `catch` clauses for the

[*try 1*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \epsilon \right\rangle}{\begin{array}{l}\left\langle \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots\ \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \\ \quad \left\langle \texttt{try}\ \textit{Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots\ \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n, \sigma', H', \epsilon \right\rangle\end{array}}$$

[*try 2*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \epsilon \right\rangle}{\begin{array}{l}\langle\ \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots\ \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n \\ \qquad \texttt{finally}\ \textit{Block}_{n+1}, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \texttt{try}\ \textit{Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots\ \texttt{E}_n\ \texttt{v}_n \\ \qquad \textit{Block}_n\ \texttt{finally}\ \textit{Block}_{n+1}, \sigma', H', \epsilon\ \rangle\end{array}}$$

[*catch 1*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \iota_j^E \right\rangle}{\begin{array}{l}\langle\ \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots \\ \qquad \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \textit{Block}_k, \sigma'', H', \epsilon\ \rangle\end{array}} \quad \begin{array}{l}\exists 1 \leq k \leq n : E \leq_w \texttt{E}_k \\ \forall 1 \leq l < k : E \not\leq_w E_l \\ \sigma'' = \sigma'[\texttt{v}_k \mapsto (l_k, \iota_j^E)]\end{array}$$

[*catch 2*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \iota_j^E \right\rangle}{\begin{array}{l}\langle\ \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots \\ \qquad \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \epsilon, \sigma', H', \iota_j^E\ \rangle\end{array}} \quad \forall 1 \leq k \leq n : E \not\leq_w E_k$$

[*finally 1*]
$$\frac{}{\begin{array}{l}\langle\ \texttt{try}\ \epsilon\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots\ \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n \\ \qquad \texttt{finally}\ \textit{Block}_{n+1}, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \textit{Block}_{n+1}, \sigma, H, \epsilon\ \rangle\end{array}}$$

[*finally 2*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \iota_j^E \right\rangle}{\begin{array}{l}\langle\ \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots \\ \qquad \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n\ \texttt{finally}\ \textit{Block}_{n+1}, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \textit{Block}_k\ \ \textit{Block}_{n+1}, \sigma'', H', \epsilon\ \rangle\end{array}} \quad \begin{array}{l}\exists 1 \leq k \leq n : E \leq_w \texttt{E}_k \\ \forall 1 \leq l < k : E \not\leq_w E_l \\ \sigma'' = \sigma'[\texttt{v}_k \mapsto (l_k, \iota_j^E)]\end{array}$$

[*finally 3*]
$$\frac{\left\langle \textit{Stmt}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma', H', \iota_j^E \right\rangle}{\begin{array}{l}\langle\ \texttt{try}\ \textit{Stmt;Stmts}\ \texttt{catch}\ \texttt{E}_1\ \texttt{v}_1\ \textit{Block}_1\ \ldots \\ \qquad \texttt{E}_n\ \texttt{v}_n\ \textit{Block}_n\ \texttt{finally}\ \textit{Block}_{n+1}, \sigma, H, \epsilon\ \rangle \\ \quad \rightsquigarrow_P \langle\ \textit{Block}_{n+1}, \sigma', H', \epsilon\ \rangle\end{array}} \quad \forall 1 \leq k \leq n : E \not\leq_w E_k$$

[*bind 4*]
$$\frac{\left\langle s, \sigma', H', \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma'', H'', \iota_i^E \right\rangle}{\left\langle \texttt{bind}\ \sigma', H'\ \texttt{in}\ s; s'; \ \texttt{end}, \sigma, H, \epsilon \right\rangle \rightsquigarrow_P \left\langle \epsilon, \sigma, H'', \iota_i^E \right\rangle}$$

Table 6.3: Extended `Bahasa` semantics for catching and propagating exceptions

first matching variable type. If we are not in a `try` block, then we propagate the exception to the calling procedure.

Like [*sequence*] from Table 3.9, the rules [*try 1*] and [*try 2*] evaluate statements that are enclosed in a `try` statement. [*catch 1*] and [*catch 2*] handle exceptions that occur when no `finally` block is present. First is checked whether there exists a `catch` block such that the exception referenced by ι_j^E can be widened to the type of variable $\mathtt{v}_k$. If such a block is found, the exception is cleared and that block is executed. Otherwise, the exception is propagated. The rules [*finally 1*] and [*finally 2*] work analogously; additionally the `final` block is executed, even if no exception is thrown. Last but not least, rule [*bind 4*] takes care of disposing a method if an exception occurs. In this case, either a `try` statement must have finished without catching the exception, or an illegal situation like an access to `null` must have occured. [GJS1996] define that in this case the method returns immediately. The calling method must either catch the exception or return itself. This process is continued until either a method handles the exception or the program is finished.

Having added exception handling to $\mathtt{Bahasa}_r$, we now need to adopt our analysis. Just like the configurations for execution of $\mathtt{Bahasa}_r$, the abstract configuration will contain exceptions. However, as the analysis computes one solution for the whole program, we associate the exceptions with labelled points. This is justified by the fact that exceptions are represented by classes. Thus, the function domain $\widehat{\mathbf{Cache}}$ is redefined to $\widehat{\mathbf{Cache}}_{Exc} : \mathbb{N} \to \mathcal{P}(\mathbf{Classes})^2$, where the first component is the *regular* value as computed by the control flow analysis from Chapter 4. In contrast to the concrete case, the abstraction is not *one* exception but a set of exceptions. This is similar to the abstractions applied to regular variables. We redefine $\widehat{\chi}$ to access the first component of the pair and define $\mathcal{E}$ to access the second.

Furthermore, method bodies are always labeled. This is necessary in order to propagate unhandled exceptions from the callee back to the caller.

We will now describe the effect of the statements on our analysis as shown in Table 6.4. Instead of describing all intermediate steps like in Chapter 4 we directly give the rules for constraint generation. One analysis is needed in addition, namely *reaching exceptions*, which is denoted by $re \in RE : \mathbb{N} \to \mathcal{P}(\mathbb{N})$. Just like rd, it computes for each label the set of labels where an exception is defined, i. e. thrown.

The first thing to handle is throwing of exceptions. As shown above, exceptions are created and initialized just like normal objects. When they are thrown, the rewriting rules store the reference to the exception object in the configuration. The effect of the abstract `throw` statement is to store the set associated with the label of the variable in the second component of $\widehat{\chi}$. Additionally, the exception must be associated with the return label of the procedure.

Care must also be taken to propagate uncaught exceptions from the return label of a method to the calling methods. This is checked by the not shown adopted rules for the control flow analysis. With respect to the constraint graph and its solution, this is assured by the semantics of the `CALL` functor. Namely, the solver not only adds an edge between n_{l_r} and n_{l_e} to let flow back the result of the method, but also between $n_{l_r^{\mathcal{E}}}$ and $n_{l_e^{\mathcal{E}}}$. Thus, thrown exceptions can be dealt with

$\widehat{[\mathit{Throw}]}_{\mathcal{C}}$	$C_\star[\text{throw } \texttt{VarName}^l] = C_\star^R[\text{VarName}, l] \cup \{\widehat{\chi}(l) \subseteq \widehat{\mathcal{E}}(l), \widehat{\mathcal{E}}(l) \subseteq \widehat{\mathcal{E}}(l_r)\}$ where l_r is the label assigned to the body of the method analyzed
$\widehat{[\mathit{try}]}_{\mathcal{C}}$	$C_\star[\text{try } Block_t \text{ catch } (\texttt{ClassName VarId}^{l_i}\ Block_i)_{1\leq i\leq n}] :=$ $C_\star[Block_t] \cup \bigcup_{1\leq i\leq n} C_\star[Block_i] \cup$ $\{\mathcal{E}(l') \subseteq \widehat{\chi}(l_i) \mid \forall 1 \leq i \leq n, l' \in re(l_i)\}$
$[\mathit{try} \cdots \widehat{\mathit{finally}}]_{\mathcal{C}}$	$C_\star[\text{try } Block_t \text{ catch } (\texttt{ClassName VarId}^{l_i}\ Block_i)_{1\leq i\leq n} \text{ finally } Block_f\ ] :=$ $C_\star[Block_t] \cup \bigcup_{1\leq i\leq n} C_\star[Block_i] \cup C_\star[Block_f] \cup$ $\{\mathcal{E}(l') \subseteq \widehat{\chi}(l_i) \mid \forall 1 \leq i \leq n, l' \in re(l_i)\}$

Table 6.4: Generating constraints for exception handling

in the calling method.

The graph constructed from a constraint set with exceptions contains two kinds of nodes – those that origin from the normal program execution and those that deal with exceptions. At `throw` and `catch` statements, there exist interfaces between these two graphs. As exceptions are modeled similar to the normal cases, the solving algorithm needs no change.

As easily can be seen, the modularity of our approach is not changed by introducing exceptions. They are mostly handled by the constraints of the method where they are thrown. Only when the solver evaluates a `CALL` functor, exceptions may flow between different methods.

Chapter 7

Related Work

Much research effort has been put in developing techniques for computing an accurate call graph for object-oriented languages.

The approaches are either context and flow insensitive, resulting in fast but imprecise solutions, or do not scale due to bad performance. In presenting the different analyses we will take the example program provided in Chapter 3. We have extended it to contain a third class `Election3`, that increases the number of votes by 3 for every ballot. The program is given in Figure 7.1. As seen before it consists of method `main` in class `App` that creates an object of class `Election` and one of `Election2`. Figure 7.4 shows the class hierarchy graph including the signatures of the methods defined.

The program contains two kinds of call sites – those to constructors and those to normal methods. Calls to constructors are not dispatched since the target method is perfectly known. Thus, the only call sites that are interesting are the two calls to the method `tick`.

In what follows we will present the different analyses. We adopt the classification technique from [TP2000], where algorithms are classified according to the number of sets they use to approximate run-time values. We give similar characterizations for all analyses.

For each approach we will present the call graph with the determined call edges between call sites and methods called. The call graph is an abstraction of the interprocedural call graph. It contains the methods and for each method `m` the call sites in `m`. Figure 7.2 shows the ideal call graph for the example program that has exactly one edge for each of the two call sites in `App.main`. We do not show the calls to constructors, as for these there is exactly one target determined at compile-time, that is, the calls generated are not dispatched.

Figure 7.3 summarizes the number of outgoing edges for the two call sites in the example program. Of course this example has been constructed to easily demonstrate the weak points of the different approaches. However, it is legitimate to suppose that in real world programs similar situations occur. The results in Chapter 8 prove that this indeed is the case.

What all approaches have in common is a set R of reachable methods, that is initialized to the

```
class Object {
    void Object(Object tg) { return; }
}
class Election extends Object{
    int count;
    void Election(Election ta) {
        ta.Object(ta);
        ta.count = 1; return;
    }
    void tick(Election tb) {
        int x;
        x = tb.count;
        x = x + 1;
        tb.count = x;
        return;
    }
}
class Election2 extends Election{
    void Election2(Election2 tc) {
        tc.Election(tc);
        return;
    }
    void tick(Election2 td) {
        int y;
        y = td.count;
        y = y + 1;
        td.count = y;
        return;
    }
}
class Election3 extends Election{
    void Election3(Election3 te) {
        te.Election(te);
        return;
    }
    void tick(Election3 tf) {
        int z;
        z = tf.count;
        z = z + 3;
        tf.count = z;
        return;
    }
}
class App extends Object{
    static void main() {
        Election candA,candB;
        candA = new Election();
        candA.Election(candA);
        candB = new Election();
        candB.Election(candB);
        candB = new Election2();
        candB.Election2(candB);
        candB.tick(candB);
        candA.tick(candA);
        return;
    }
}
```

Figure 7.1: Extended example program

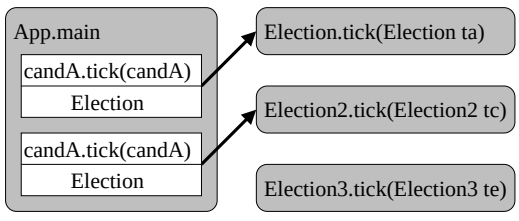


Figure 7.2: The correct call graph

Analysis	call site 1	call site 2
no analysis	3	3
CHA	3	3
RTA	2	2
VTA	1	2
DTA	2	2
XTA	2	2
CFA	1	1

Figure 7.3: Number of call edges for the example program.

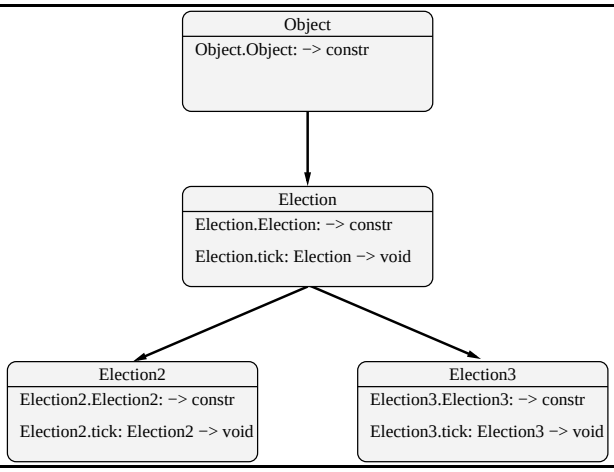


Figure 7.4: Class hierarchy graph for the example program

entry points of a program.

7.1 Name Based Resolution

Name based resolution (NBR) is a simple algorithm that exists in two variations. The one kind only takes the *name* of methods into account for constructing the call graph. The other kind requires the *signature* of methods to be equal. The simpler name based version can be described by

- `p` $\in R$ for all entry points `p` of a program
- for each method `M` containing a virtual call site `e.m(···)` and each method `M′` with name `m`: `M` $\in R \Rightarrow$ `M′` $\in R$.

These constraints state that the entry points of a program are reachable and that for each call site in a reachable method all methods with matching names are reachable. The set R we are interested in is the least set fulfilling these constraints. This allows to delete as many methods as possible.

For the example program, name based resolution cannot distinguish between the different definition of method `tick`. Thus for the two call sites each of the implementations occurs to be callable. The result is illustrated in Figure 7.5.

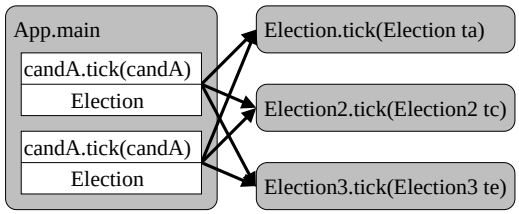


Figure 7.5: Call graph constructed by Name Based Resolution and Class Hierarchy Analysis

7.2 Class Hierarchy Analysis

Class hierarchy analysis (CHA) [DGC1995, Fer1995] provides a slight improvement compared to name based resolution. Beside the name of a method it also takes into account the defined type of an expression. Just like NBR, class hierarchy analysis only uses one set R for the whole program.

- $\mathtt{p} \in R$ for all entry points $\mathtt{p}$ of a program
- for every method $\mathtt{M}$, every call site $\mathtt{v.[AT]m(\cdots)}$ in $\mathtt{M}$ and each $\mathtt{C'} \leq_w \Gamma_V(\mathtt{v})$ with $\mathit{MostSpecial}(\mathtt{m}, \mathtt{C'}, \mathit{AT}) = \{(\mathtt{C''}, \mathit{MT}, \mathtt{mb})\} \equiv \mathtt{M'}$:
  $\mathtt{M} \in R \Rightarrow \mathtt{M'} \in R$

The first constraint again demands all entry points of a program to be reachable. The second constraint is a little bit more restrictive than the second constraint of NBR – in addition it requires the method to be visible in a sub class $\mathtt{C'}$ of the class of $\mathtt{v}$.

Also this approach is not able to limit the number of call edges, as the declared types of the variables `candA` and `candB` is `Election`. Thus any of the implementations of `tick` could be called. Thus as illustrated in Figure 7.5, the call graph constructed based on CHA is the same as for NBR.

7.3 Rapid Type Analysis

Rapid type analysis (RTA) [Bac1998] has been developed as an extension of CHA. Like that, it takes into account the declared types of the expressions that are used to call methods. Additionally, RTA collects information, objects of which classes may be created. That is, all `new` statements are collected in a newly introduced set C. The constraints are

- for all entry points $\mathtt{p}$ of a program
  $\mathtt{p} \in R$

- for every method `M`, every call site `v.[AT]m(···)` in `M` and each `C` $\leq_w$ $\Gamma_V(\mathtt{v})$ with *MostSpecial*$(\mathtt{m}, \mathtt{C}, AT) = \{(\mathtt{C}', MT, \mathtt{mb})\} \equiv \mathtt{M}'$:
  $\mathtt{M} \in R \wedge \mathtt{C} \in S \Rightarrow \mathtt{M}' \in R$
- for every method `M` and every object creation `new C()` in `M`:
  $\mathtt{M} \in R \Rightarrow \mathtt{C} \in S$

As can be seen the constraints are almost equal to that of CHA. The extension is, that the method selection in the second constraint is restricted to match only those methods that are members of classes that really are created at run-time.

As the only `new` statements in the program create objects of class `Election` and class `Election2`, the implementation of `tick` in class `Election3` is detected to be uncallable.

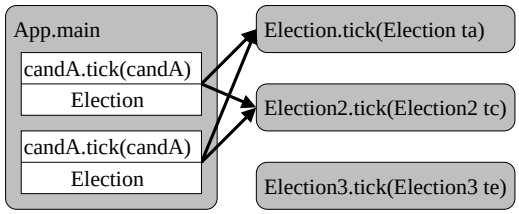


Figure 7.6: Call graph constructed by Rapid Type Analysis, Declared Type Analysis and Extended Type Analysis

7.4 Variable Type Analysis

Rapid type analysis still is rather coarse grained in a sense that it claims that an object with declared type T may at run-time point to an object of type T' with $T \leq_w T'$ whenever there is an instantiation of T' anywhere in the program. Variable type analysis (VTA) [SHR+2000] not only inspects the `new()` statements but also looks at assignments between variables. The requirements imposed on the form of programs are similar to those we formulated in Chapter 3.

The analysis is based on a *type propagation graph*. Each of the nodes represents a local variable, a method parameter or a class member that has class type. Just like the labels we assign to method bodies, there is a special node for methods that return a class value. The edges in the graph resemble assignments between the entities represented by nodes. For method calls edges are added between arguments and parameters just as seen in Chapter 4. However, as VTA is a *flow insensitive* analysis, the targets of call sites must be approximated statically. Here, a standard technique like RTA or CHA is used.

VTA is performed in four steps:

1. construct a conservative call graph

2. build the type propagation graph
3. initialize those nodes with $\{T\}$ that represent variables that are assigned a newly created object of type T
4. propagate this information in order to compute a fixpoint of sets of classes

The first two steps can be combined to be performed at the same time. Again, this can be modeled by representing the nodes in the type propagation graph as sets.

Then, the constraints again look similar to those already seen:

- for all entry points p of a program
  $\mathtt{p} \in R$
- for every method M, every call site $\mathtt{v.[AT]m(v_1, \cdots, v_n)}$ in M and each $\mathtt{C} \leq_w \Gamma_V(\mathtt{v})$ with $\mathit{MostSpecial}(\mathtt{m}, \mathtt{C}, AT) = \{(\mathtt{C}', MT, \mathtt{mb})\} \equiv \mathtt{RT\ M'(PT_1\ p_1, \cdots, PT_n\ p_n)}$:
  $\mathtt{M} \in R \wedge \mathtt{C} \in S \Rightarrow \mathtt{M}' \in R \wedge \mathtt{n}_{v_i} \subseteq \mathtt{n}_{p_i}$
- for every method M and every object creation `new C()` in M:
  $\mathtt{M} \in R \Rightarrow \mathtt{C} \in S$
- for every method M and every assignment between two entities represented as nodes n_{lhs} and n_{rhs}:
  $\mathtt{M} \in R \Rightarrow \mathtt{M}_{n_{rhs}} \subseteq \mathtt{M}_{n_{lhs}}$

Having constructed the type propagation graph, its strongly connected components are identified and collapsed. As the resulting graph is static and will not change anymore, algorithms for union based problems like that presented in [WM1995] can be used to compute the sets in one pass over the graph.

It is noteworthy, that the overall complexity of the total analysis mainly depends on that of the analysis used for computing the approximative call graph.

The type propagation graph used to construct the call graph is shown in Figure 7.8. The nodes represent variables where the upper half contains the variable name in the notation

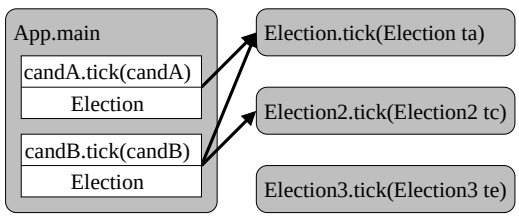


Figure 7.7: Call graph constructed by Variable Type Analysis

`className.methodName.variableName`. The lower half contains the computed set. The resulting call graph is given in Figure 7.7. As can be seen, VTA has found out that the created object of `Election2` will not be stored in `candA`.

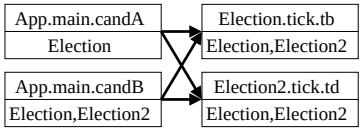


Figure 7.8: Type propagation graph for VTA

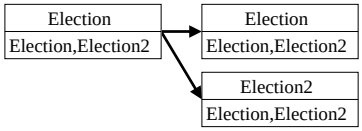


Figure 7.9: Type propagation graph for DTA

7.4.1 Declared Type Analysis

In [SHR+2000] the authors also introduce a second technique called declared type analysis which is far less precise then VTA. Instead of mapping each variable to a node of the type propagation graph, now a set is associated with each declared type. The resulting graph is given in Figure 7.9. As one would expect, the result given in figure 7.6 is less precise then with VTA, namely it equals the one created by RTA. In the example this is due to the fact that the two calls are based on variable with the same declared type. The advantage of DTA is that the resulting type propagation graph has fewer nodes and edges. However, as the graph resulting from VTA can be solved in one pass it does not seem to be necessary to choose DTA.

For reasons of completeness we give the constraints. Again, implicitly RTA is used.

- for all entry points $\mathtt{p}$ of a program
  $\mathtt{p} \in R$
- for every method $\mathtt{M}$, every call site $\mathtt{v.[AT]m(v_1, \cdots, v_n)}$ in $\mathtt{M}$ and each $\mathtt{C} \leq_w \Gamma_V(\mathtt{v})$ with $\mathit{MostSpecial}(\mathtt{m}, \mathtt{C}, AT) = \{(\mathtt{C}', MT, \mathtt{mb})\} \equiv \mathtt{RT\ M'(PT_1\ p_1, \cdots, PT_n\ p_n)}$:
  $\mathtt{M} \in R \wedge \mathtt{C} \in S \Rightarrow \mathtt{M}' \in R \wedge \mathtt{n}_{\Gamma_V v_i} \subseteq \mathtt{n}_{\Gamma_V p_i}$
- for every method $\mathtt{M}$ and every object creation `new C()` in $\mathtt{M}$:
  $\mathtt{M} \in R \Rightarrow \mathtt{C} \in S$
- for every method $\mathtt{M}$ and every assignment between two entities represented with static type T_{lhs} and T_{rhs}:
  $\mathtt{M} \in R \Rightarrow \mathtt{M}_{T_{rhs}} \subseteq \mathtt{M}_{T_{lhs}}$

As can be seen the constraints are structurally equivalent with those from VTA. The difference is that all variables and fields with the same declared type are mapped onto the same node in the type propagation graph.

7.5 Extended Type Analysis

The algorithms presented up to now can be characterized by an increase in sets used to abstract the program. Extended type analysis (XTA) as presented by [TP2000] uses a set for each method and each field. These sets are used to propagate sets of created classes over call edges. The idea is, that objects created in function m are also visible in functions m' that are called from m. Equivalently, if method m' has a class return type, objects created in m' may flow back to m.

The constraints for XTA are formulated by means of a function $subtypes : \mathcal{P}(\textbf{Classes}) \rightarrow \mathcal{P}(\textbf{Classes})$, $subtypes(S) = \{C | C \leq_w C' \wedge C' \in S\}$.

- for all entry points `p` of a program
  $\texttt{p} \in R$
- for every method `M`, every call site `v.[AT]m(v`$_1$`, ···, v`$_n$`)` in `M` and each `C` $\leq_w \Gamma_V(\texttt{v})$ with *MostSpecial*$(\texttt{m}, \texttt{C}, AT) = \{(\texttt{C}', MT, \texttt{mb})\} \equiv \texttt{M}'$:
  $(\texttt{M} \in R \wedge \texttt{C} \in S_M) \Rightarrow \texttt{M}' \in R \wedge subtypes(AT) \cap S_M \subseteq S_{M'} \wedge subtypes(AT) \cap S_{M'} \subseteq S_M \wedge C \in S_{M'}$
- for every method `M` and every object creation `new C()` in `M`:
  $\texttt{M} \in R \Rightarrow \texttt{C} \in S_M$
- for every method `M` and every read of field `x`:
  $\texttt{M} \in R \Rightarrow S_x \subseteq S_M$
- for every method `M` and every write `v.[C]x`:
  $\texttt{M} \in R \Rightarrow subtypes(\textit{FieldDecl}(C, x)) \cap S_M \subseteq S_x$

The resulting constraint graph ensures, that each method is analyzed with the sets of classes of objects that may reach the method. This is modeled by passing sets of classes along call edges and back along return edges. Like in our analysis, each class or instance field is modeled by a

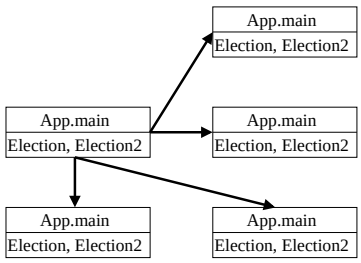


Figure 7.10: Type propagation graph for Extended Type Analysis

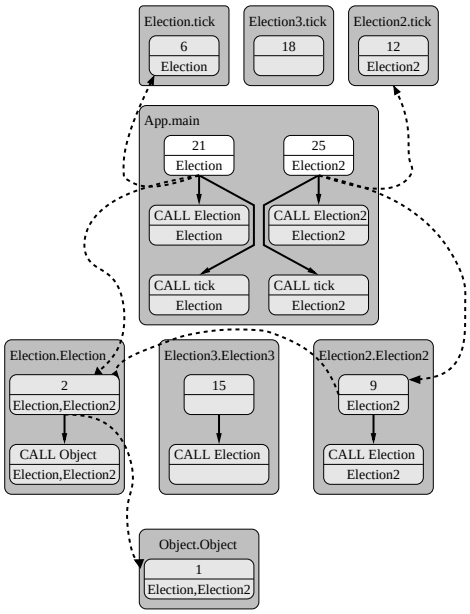


Figure 7.11: Constraint graph for 0-CFA

single set. Whenever a read from or a write to a field `f` occurs in method `m`, the set S_{f} is added to S_{m}.

Figure 7.10 shows the constraint graph for the example program. The upper string is the name of the method `M` represented by the node, the lower strings represent the classnames in S_{M}.

Compared to VTA, XTA does not associate single sets with variables. This is the reason, that for the two call sites the implementations of `tick` in `Election` and `Election2` are detected to be callable. Thus, the call graph equals that computed by RTA and DTA (c. f. Figure 7.6).

7.6 Control Flow Analysis

Control flow analysis originally has been developed for the analysis of functional programming languages [Shi1991, NNH1999]. A typical feature of imperative languages is, that each node only has a small number of successors. As this fails for higher–order languages, control flow analysis tries to identify for each expression the set of possible functions it may evaluate to at run-time. An introduction can be found in Section 2.1 and Chapter 4.

As described in Chapter 4, CFA labels all interesting program points. Actually, as the example used in this chapter almost equals that from Chapter 4, we will not repeat the construction of the constraint graph. Figure 7.11 gives the already solved graph. As seen before, special nodes have been colored. The resulting call graph is the correct one containing exactly one edge per call site

as given in Figure 7.2.

The constraint formulation is rather complex as seen in Chapter 4. This is caused by the complexity of the analysis. While the approaches seen up to new are flow and context insensitive, CFA is not. That is, it actually needs to model the flow of data in procedures and thus needs to inspect every single statement. This has been shown in Tables 4.7 and 4.8. In our approach, the set R has been modeled in the solver by the set $called$.

Chapter 8

Evaluation and Applications

This chapter provides an evaluation of rapid control flow analysis. RCFA has been implemented in `JoC`, the `JOSES` compiler [RBLP2000]. In order to allow for a comparison with standard techniques, additionally rapid type analysis has been implemented, c. f. [Bac1998] and Section 7.3. These analyses will be applied to real world programs.

One problem occurring due to the run-time framework developed in the project is that exceptions are not supported. While they easily fit into our analysis as shown in Chapter 6, their implementation in the run-time system has not yet been finished. Unfortunately, exceptions are a heavily used mechanism for execution control in most object-oriented languages. Some researchers even promote them as a standard approach to control program execution [Hor2000]. Of course, the choice of benchmarks has been heavily influenced by this limitation. However, as [TP2000] point out, while the effect of exceptions on program execution is important its impact on the complexity of call graph construction is minimal.

Additionally, the run-time system is based on a restricted set of the Java standard classes. Features like *dynamic class loading* and *reflection* have been cut out in order to allow application of static analyses. The first one allows to dynamically load classes into a running program. As a result, the program may create objects of this new classes and may call objects on them. That is, the set of created classes may be changed at run-time. However, all optimizations performed require this set to be static. There has been some work on how to combine these features with optimizations, c. f. [ST2000]. This approach may be adopted to a framework using our analysis. *Reflection* also is problematic, as it allows the program to determine methods available from a class. It thus may dynamically construct a method call and execute it. However, this makes it impossible to predict and limit the set of callable methods.

We will now present the evaluation results. Section 8.2 will demonstrate some practical applications of the results of the analysis.

All measurements have been taken on a `DELL Latitude C600` with a 750MHz Pentium III and 384MB main memory.

benchmark	classes #	methods #	stmts #	calls static	calls dispatched
HelloWorld.java	209	1629	26511	1615	1943
threads.java	232	1822	30614	1985	2342
methodbench.java	251	2043	37739	2718	3013
compress.java	265	2128	38132	2436	2972

Table 8.1: Static characteristics of the benchmark programs

benchmark	sccs #	edges #	LOAD #	STORE #	CALL #
HelloWorld.java	948	695	57	52	207
threads.java	1877	1438	108	87	444
compress.java	2150	1701	193	91	490
methodbench.java	2855	2139	164	91	732

Table 8.2: Static characteristics of the generated constraint graphs

8.1 Evaluation

Before giving the evaluation results we first give a short overview of the benchmarks used and the criteria evaluated.

8.1.1 Benchmark Programs

`HelloWorld.java`

The first benchmark we use is `HelloWorld.java`. This may sound astonishing, as usually translating and analyzing this program is a typical beginner's test case. However, recall that we analyze *whole* programs. Thus we need to handle the standard Java libraries which makes it quite a step to be able to handle the one liner.

`compress.java` A Java implementation of the well known file compression program `compress` serves as the second benchmark. We will run the generated binary on input files with different sizes. This allows to get a feeling for the impact of the optimization on the dynamic behavior of a program.

`threads.java` The third benchmark program serves to demonstrate that our analysis is thread aware. Using techniques like those presented in [CKRW1998, CKRW1999],

`Bahasa` semantics could easily be extended to handle threads. Due to the abstraction applied to class and instance members, the analysis can handle threads anyway.

`JavaGrande, JGFMethodBench` This is a part of the JavaGrande benchmark suite. It tests method calling on several constructs. For this benchmark we give the data reported by the benchmark. These cannot be compared to results found in literature. As already pointed out, our framework lacks some important features compared to others.

Tables 8.1 and 8.2 give an overview of some static characteristics of our benchmarks. These are

- the number of classes and methods in the class, including those from the standard libraries
- the number of call statements in the methods, split into static and dynamic calls
- the size of the generated constraint graph
- the size of the solved constraint graph

The last three include the counts of `LOAD`, `STORE` and `CALL` functor nodes in the graph. These numbers are given for normal and optimized total graph and for the solved graph of the modular approach.

8.1.2 Evaluation Criteria

The evaluation is based on several criteria, that are typically used to compare techniques for construction of the interprocedural control flow graph.

Analysis Result. The result is measured by the count of

- types available per method and per program
- monomorphic and polymorphic call sites
- call edges in the resulting graph
- reachable methods

The number of available types per method can be considered the essential measure for call graph construction algorithms. The number of reachable methods and the number of call edges directly depends on this count.
Monomorphic call sites are those that have only one possible target. Equivalently, *polymorphic* call sites have two or more possible targets. This classification is of particular interest, as for monomorphic call sites the constructed call graph is exact.
The number of reachable methods is a measure for the reduction of the resulting program size.

Analysis and Optimization Time. This is the total duration of the analysis and the optimization, that is including the collection of data from the intermediate representation, analyzing it and optimizing the IR.

Total Compilation Time. We measure the total run-time of `JoC`. The time is measured for a compiler run without any further optimizations. That is, the program is parsed, the analysis is applied, the object-oriented extensions are lowered and code is generated.

Binary Size. The size of the binary created. This is directly related to the number of reached methods and is given for reasons of completeness.

Dynamic Counts. We run the created binary and count for each call how often it is executed. The distinguished calls are

- static calls to class members
- polymorphic dynamic calls
- monomorphic dynamic calls

8.1.3 Results

Table 8.3 gives the results of running RTA and RCFA on our benchmark programs. The first line gives the relevant numbers for running `JoC` without any optimization. For each benchmarks the following lines give the results for running RTA and RCFA. For RCFA three lines are reserved - RCFA gives the times for recompiling the constraints. In measuring *precompiled*, the precompiled constraints have been used. Finally, 1-RCFA gives the results for running a 1-CFA.

There are several interesting points regarding this table. First, the number of reachable methods is significantly reduced between 16% and 40%. However, what is even more important is the number of call edges in the resulting call graph. This is reduced to 41% to 69%. This is a direct consequence of the flow sensitiveness of rapid control flow analysis. The most interesting results are the numbers for 1-RCFA. While the expectation would be to gain precision when adding contexts, this obviously is not the case due to the abstraction we use in our solver. Since we construct the graph consisting of the strongly connected components, we can not perform destructive update on propagated sets. Thus, even if we have several copies of a method in our constraint graph, still the set of all classes computed for the object is propagated.

Table 8.4 gives a detailed overview of the classifications of call sites. We give the percentage of unreached, monomorphic and polymorphic call sites remaining in the program. Additionally, the changes from RTA to RCFA are given, that is how RCFA improves on the classification as performed by RTA. It is noteworthy that already RTA does a very good job in classifying monomorphic call sites.

Finally, Table 8.6 gives an overview of some execution counts. We have instrumented the generated code to sum executions of *static*, *dispatched* and *resolved* calls, respectively. These numbers

benchmark	classes #	procedures #	analysis time	compile time	rounds #	call edges #	setsize total	setsize per method
HelloWorld	209	1629		214.2				
RTA	96	200	4.60	109.6		625	50	
RCFA	73	159	5.41	103.7	887	352	44	2.79
precompiled	73	159	4.65	102.9	887	352	44	2.79
1-RCFA	73	159	4.69	102.9	1325	352	44	2.79
threads	232	1822		290.7				
RTA	123	370	7.3	173.6		1775	92	
RCFA	89	233	7.8	149.6	1965	735	63	2.91
precompiled	89	233	6.4	147.1	1965	735	63	2.91
1-RCFA	89	233	6.5	147.3	3695	735	63	2.91
methodbench	251	2043		384.1				
RTA	136	421	11.2	227.0		2707	95	
RCFA	98	254	13.7	199.2	2948	1166	67	3.02
precompiled	98	254	10.7	195.6	2948	1166	67	3.02
1-RCFA	98	254	11.7	196.6	8202	1166	67	3.02
compress	265	2128		412.4				
RTA	129	329	11.0	222.2		1125	84	
RCFA	102	278	12.7	210.0	2407	787	67	3.05
precompiled	102	278	11.5	208.7	2407	787	67	3.05
1-RCFA	102	278	11.9	209.1	4865	787	67	3.05

Table 8.3: Results for RCFA and RTA

benchmark	RTA unreached	RTA mono	RTA poly	RCFA unreached	RCFA mono	RCFA poly
helloworld	95.4%	4.0%	0.6%	97.1%	2.8%	0.1%
threads	89.7%	8.5%	1.8%	94.5%	5.4%	0.1%
methodbench	88.3%	9.7%	2.0%	93.2%	6.8%	0.1%
compress	92.6%	6.5%	0.9%	94.0%	5.8%	0.2%

Table 8.4: Classification of dispatched call sites

are given for the unoptimized benchmarks and for those optimized with RTA and RCFA. The compress benchmark has been run on one data file of 1 kByte size for one round and on a 2 kByte big file. What can be seen is that obviously all input size related calls could be resolved by RTA and RCFA.

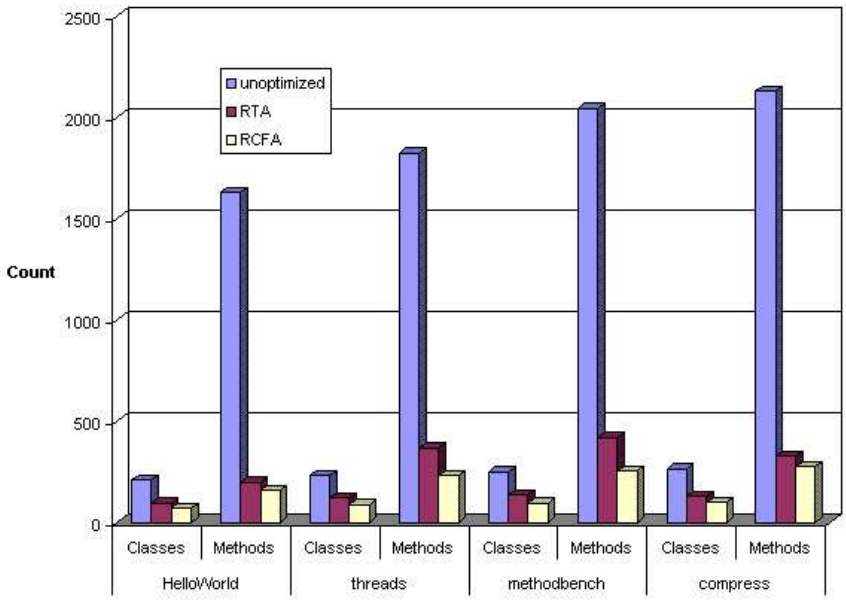


Figure 8.1: Improvement on class and method counts

benchmark	mono→		poly→		
	unreached	mono	unreached	mono	poly
helloworld	77	158	23	7	7
threads	212	351	104	8	7
methodbench	249	522	134	14	7
compress	64	388	33	11	14

Table 8.5: Changes in the classification of dispatched call sites

benchmark	unoptimized			RTA			RCFA		
	static	disp.	res.	static	disp.	res.	static	disp.	res.
helloworld	173	5944	0	173	18	5926	173	10	5934
threads	468	6628	0	468	35	6593	468	20	6608
methodbench	2301	7923	0	2301	270	7653	2301	96	7827
compress 1KB	672	17494	0	672	108	17386	672	74	17420
compress 2KB	672	27605	0	672	108	27497	672	74	27531

Table 8.6: Execution counts for call statements

benchmark	unoptimized		RTA		RCFA	
	perf.	del.	perf.	del.	perf.	del.
helloworld	5926	0	32	5894	30	5896
threads	6585	0	66	6519	60	6525
methodbench	33127	0	715	32412	512	28615
compress	7356	0	85	7271	77	7279

Table 8.7: Execution counts for null pointer checks

8.2 Practical Applications

Having shown the evaluation data for rapid control flow analysis, we now turn to presenting some practical applications of the results. One of our claims is that static analysis benefits from the increased accuracy of the interprocedural control flow graph. We will present results for such an analysis, the elimination of null pointer checks. It has been implemented using the program analyzer generator `PAG`, c. f. [Mar1999b].

Finally, we briefly detour to Eiffel, another statically typed object-oriented language. Some years ago, there has been a discussion regarding a supposed type error in its type system, c. f. [Mey1988, Coo1988]. We will show that rapid control flow analysis could have helped in resolving the problem.

8.2.1 Eliminating Null Pointer Checks

When introducing exceptions in Chapter 6 we mentioned that for every access to an object the Java language definition [GJS1996] requires that the object is checked to be not `null`. If it happens to be `null`, a null pointer exception must be thrown. Obviously, this is a very pessimistic approach. E. g. when a dynamically dispatched method is called, the object on that the call was executed has been checked already. When this object is in the method to access some class members or call other methods, clearly no further check should be performed.

The analysis and optimization we have implemented traces already checked variables through the program. The generated code is instrumented to count necessary and resolved null pointer checks. Tables 8.7 and 8.8 give the results; these include the analysis effort, that is the steps performed by the PAG analyzer, and the run-time counts of performed and resolved null pointer checks. As can be seen in Figure 8.2, the number of steps needed by the analyzer grows exponential with the number of call edges in the call graph. Thus, a call graph as exact as possible is worth the higher effort needed to compute a flow-sensitive solution.

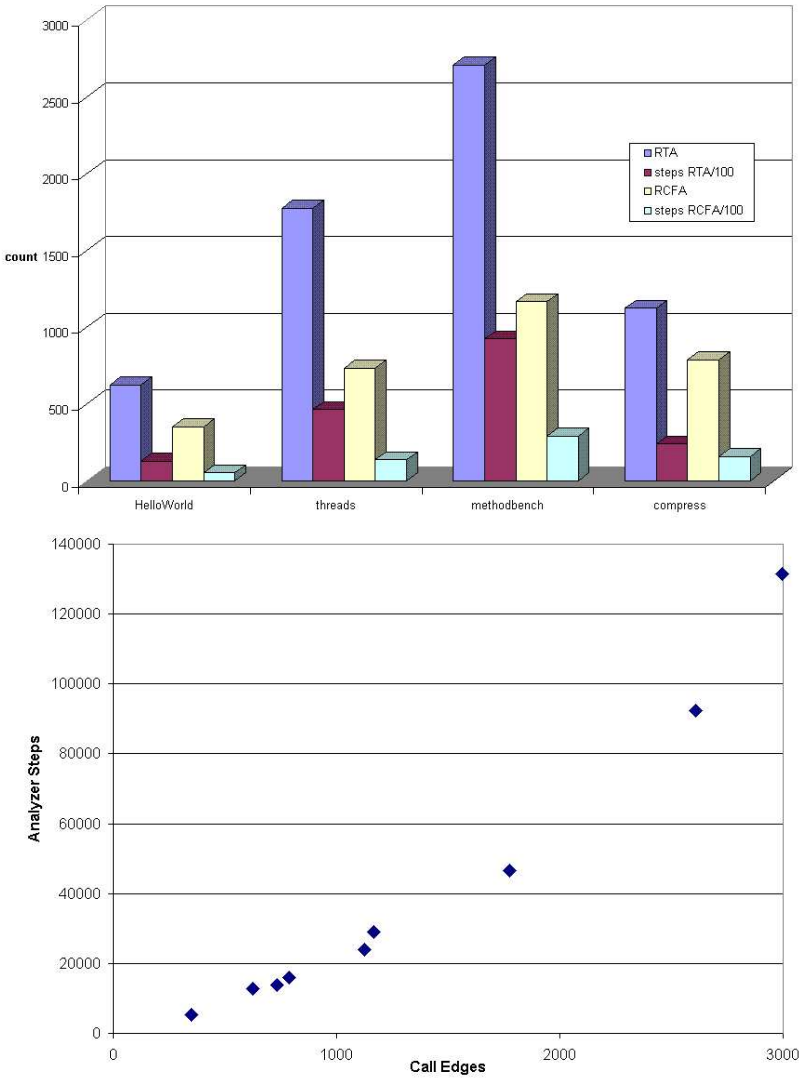


Figure 8.2: Relation between number of call edges and analyzer steps

benchmark	RTA + NPC		RCFA + NIL	
	edges	steps	edges	steps
helloworld	625	12749	352	5195
threads	1775	46487	735	13844
methodbench	2707	92480	1166	28986
compress	1125	23947	787	15969

Table 8.8: Call edges and analyzer performance

8.2.2 Eiffel Type Error

Eiffel [Mey1988] is a statically typed object-oriented language, that is, it belongs to the same language class as Java. In the late eighties there was a discussion regarding a loophole in the Eiffel type system [Coo1988, Mey1989].

This loophole was introduced because in Eiffel classes may not only inherit, overwrite or define methods; they also may *remove* methods from their interface. Assume classes `P`, `H`, where `P` defines a method `f` (what Eiffel calls features), and `H` extends `P` but removes `f` from its interface. One instance of the type loophole phenomenon can be constructed by

```
p:  P; h:  H;

h.Create;
p := h;
p.f;
```

This will result in a run-time error, since `p` will point to a `H` object. While the static type of `p` knows about feature `f`, the run-time type does not. Using flow-sensitive techniques, the compiler would be able to detect at least some of these errors.

In order to prohibit this kind of run-time errors, the Java designers do not allow a child class to define a method with more restrictive attributes than the parent class. That is, in Java terms `H` could not declare method `f` as private.

In [Mey1989] an incremental type checker is proposed, that shall prohibit this type loopholes. Actually, its functionality is a side effect of rapid control flow analysis.

Chapter 9

Conclusion

If you permit yourself to read meanings into (rather than drawing meanings out of) the evidence, you can draw any conclusion you like.

Michael Keith, "The Bar-Code Beast", The Skeptical Enquirer Vol 12 No 4 p 416

In the previous chapters, we have presented a system for flow-sensitive, scalable interprocedural control-flow graph construction, using a demand-driven scalable solver for constraint based control flow analysis. Construction of the ICFG is a key problem when analyzing, optimizing and translating object-oriented languages.

The problem occurs due to the concept of *dynamic dispatch*, where only at run-time is decided which implementation of a method to call. Thus, the compiler may not easily determine possible targets of method calls. Here, techniques for construction of the ICFG help.

On the one hand, they enable to identify methods that may never be called. Since object-oriented languages usually use huge standard libraries, this is important when whole-program analysis is performed. As seen in the evaluation, many methods can be deleted as they can be identified as unreachable.

On the other hand, having constructed the ICFG the large set of well-known interprocedural data flow analyses that have been developed for imperative languages may be applied. These analyses require a fixed control-flow graph that usually does not exist in a compiler for object-oriented languages.

This chapter gives an outlook on possible future development and concludes the achievements.

9.1 Outlook

The framework we used to implement our analysis seems to be quite powerful. We were able to implement a 1-CFA using the same formalism as presented in this thesis. Due to time constraints we could not further investigate the possibilities of our framework. This remains to be done.

The solver we use is based on strongly connected components. These do not directly allow destructive update, e. g., in the case of 1-CFA. Admittedly this is suboptimal with respect to precision, but did seem acceptable in order to allow a scaling approach. It has to investigated further, whether destructive update can be added while at the same time keeping scalability.

The major hindrance for our work has been the run-time system. As soon as exceptions will be supported, we will be able to investigate the behavior of our approach using these. Due to the separation of the solution for *normal* and *exceptional* data, it should be expected that the result will be promising.

Last but not least we plan to reconsider the process of inserting edges into the graph. The strong connectedness of the graph and the topological ordering are properties that should be kept up to date dynamically. Our approach is fast, but in the worst case it revisits the whole subgraph the newly inserted edge belongs to.

9.2 Achievements

We have presented an implemented approach to construction of the ICFG. We have taken existing techniques and have combined them to a scaling approach. A standard Java semantics allowed us to prove the correctness of our control flow analysis (CFA). The CFA has been transformed to generate a constraint system. In standard approaches, this constraint system contains *all* constraints needed for analyzing the whole program. By introducing functor constraints, we are able to partition the graph into independent regions, where each region represents the constraints generated for one method. Using this independence, we are able to pre-analyze the Java standard libraries. This allows speed up of the analysis phase, as the constraint generation may be skipped. The functor constraints are handled by our solver, that evaluates them and, if necessary, adds new edges to the graph.

The solver has been applied to real-world programs to show applicability and scalability. The results are twofold.

On the one hand, the generated ICFG is significantly smaller than those generated with currently widespread techniques which we briefly review in the thesis. These techniques are discussed.

On the other hand, data flow analyzers are sped up enormously since the resulting call graph has significantly fewer call edges. We exemplarily apply an analysis for elimination of null pointer checks to our evaluation programs. Null pointer checks are one of the major reasons for bad performance of Java programs, since at every access to an object it must be checked that the object is not null. The results for this analysis show a direct correlation between the number of

call edges in the ICFG and the run-time of data flow analyzers. The results show that the number of steps performed by the analyzer grows exponential with the number of call edges in the ICFG. The extra effort needed to apply a flow-sensitive like RCFA is not a waste of resources.

This thesis proves that applying flow-sensitive techniques for construction of the ICFG are worth the effort. Due to the sensitiveness the number of call edges in the ICFG can be reduced dramatically. This allows data flow analyzers not only to run faster but also to compute better results. Additionally, during construction much more methods are detected to be unreachable than with standard, flow-insensitive techniques. The reduced number of methods and the better performance of data flow analyzers combine to create fast running binaries with small footprints.

Appendix A

Syntax of `Bahasa`

Program	→	*InterfaceList ClassList*
ClassList	→	*ClassBody ClassList* | ϵ
ClassBody	→	`class ClassId` *Extends Implements* `{` *MemberDecls MethodList* `}`
InterfaceList	→	*InterfaceBody InterfaceList* | ϵ
InterfaceBody	→	`interface InterfaceId` *InterfaceExtends* `{` *InterfaceMethodList* `}`
InterfaceMethodList	→	*InterfaceMethod InterfaceMethodList*
InterfaceMethod	→	*ReturnType* `InterfaceMethodId (` *ParList* `);`
Extends	→	`extends ClassName` | ϵ
Implements	→	`implements InterfaceName` *InterfaceNames* | ϵ
InterfaceNames	→	`,` `InterfaceName` *InterfaceNames* | ϵ
MemberDecls	→	*MemberDecl*; *MemberDecls* | ϵ
MemberDecl	→	*VarType* `MemberName`
VarType	→	`ClassName` | `InterfaceName` | *ArrayType*
ArrayType	→	`ClassName[]` | *ArrayType*`[]`
MethodList	→	*MethodBody MethodList* | ϵ
MethodBody	→	*ReturnType* `MethodId (` *ParList* `)` `{` *VarDecls Stmts* `return [` *Var* `];` `}`
ReturnType	→	`void` | *VarType*
ParList	→	*ParamList* | ϵ
ParamList	→	*Param*`,` *ParamList* | *Param*
Param	→	*VarType* `ParamName`

Table A.1: Syntax of `Bahasa`

VarDecls	→	*VarDecl* ; *VarDecls* | ϵ
VarDecl	→	*VarType* `VarName`
Block	→	{ *Stmts* } | *Stmt*
Stmts	→	*Stmt* ; *Stmts* | ϵ
Stmt	→	`if` *Expr* `then` *Block* `else` *Block*
	|	*Var* `=` *Rhs*
	|	*Call*
	|	`throw` *Variable*
	|	`try` *Block CatchList*
CatchList	→	*Catch Catches* | *Catches* `finally` *Block*
Catches	→	*Catch Catches* | ϵ
Catch	→	`catch ClassName VarName` *Block*
Rhs	→	*Var*
	|	`new ClassName()`
	|	*Call*
Call	→	*Variable*`.MethodName (` *ArgList* `)`
ArgList	→	*VarList* | ϵ
VarList	→	*Variable* , *VarList*
	|	*Variable*
Var	→	*Variable*
	|	*Member*
Variable	→	`VarName`
	|	`this`
Member	→	`ClassName.MemberName`
	|	*Variable*`.MemberName`

Table A.2: Syntax in `Bahasa` (cont.)

Appendix B

Precompiled Constraint Files Format

constraintfile	→	*header entries szkinfo*
header	→	PROC *name label* LABELS *label* (*optlabellist*)
entries	→	*entry entries*
entry	→	*node* | *classnode* | *callnode* | *loadnode* | *storenode* | *edge*
node	→	N *label label*
classnode	→	CLASS *label label name*
callnode	→	CALL *label name label* (*optlabellist*) *name name*
loadnode	→	LOAD *label name label label name*
storenode	→	STORE *label name label label name*
edge	→	EDGE *label label*
optlabellist	→	*labellist* | ϵ
labellist	→	*label labellist* | *label*
name	→	NAME
label	→	INTEGER
szkinfo	→	*szklist szksuccist*
szklist	→	*szkentry szklist* | ϵ
szkentry	→	*szk szknodes*
szk	→	SZK *label*
szknodes	→	*label szknodes* | ϵ
szksuccist	→	*szksucc szksuccist* | ϵ
szksucc	→	SZKSUCCS label *labellist*

Table B.1: Grammar for precompiled constraints files

Bibliography

[AAvS1994] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy compiler phase embedding with the cosy compiler model. In *Proceedings of the Conference on Compiler Construction CC*, 1994.

[ACE] ACE Compiler Experts b.v., Amsterdam. `www.ace.nl`.

[Acz1977] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

[Age1995] Ole Agesen. The cartesian product algorithm: Simple and precise typing of parametric polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

[AH1996] Gerald Aigner and Urs H¨olzle. Eliminating virtual function calls in C++ programs. *Lecture Notes in Computer Science*, 1098:142–??, 1996.

[AM1995] Martin Alt and Florian Martin. Generation of Efficient Interprocedural Analyzers with PAG. In Alan Mycroft, editor, *SAS'95, Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 33–50, 1995.

[Aßm1996] Uwe Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In Tibor Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 121–135, 1996.

[Aßm1998] Uwe Aßmann. A tutorial for optimix. Technical Report iratr-1998-14, University of Karlsruhe, Germany, Computer Science, 1998.

[Aßm2000] Uwe Aßmann. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):583–637, 2000.

[Aßm2001] Uwe Aßmann. Proceedings of JOSES – Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001. Technical Report 2001-10, Fakult¨at f¨ur Informatik, Universit¨at Karlsruhe, April 2001.

[Bac1998] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, 1998.

[Bau2001] J¨org Bauer. A control-flow-analysis for multi-threaded java with security applications. Master's thesis, Universit¨at des Saarlandes, 2001.

[BH1998] Bernhard Bauer and Riitta H¨ollerer. *Übersetzung objektorientierter Programmiersprachen*. Springer, 1998.

[CC1977] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CKRW1998] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded Java – an event-based structural operational semantics. *Lecture Notes in Computer Science*, 1349, 1998.

[CKRW1999] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded Java. *Lecture Notes in Computer Science*, 1523, 1999.

[CLR1992] Thomas H. Cormen, Charles Eric Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1992.

[Coo1988] William Cook. A proposal for making eiffel type-safe. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1988.

[DE1997] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, 1997.

[DE1999] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. *Lecture Notes in Computer Science*, 1523, 1999.

[DGC1995] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952, 1995.

[Die1996] Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Universit¨at des Saarlandes, 1996.

[DP2001] Holger Dewes and Christian W. Probst. Static method call in Java. *Proceedings of the Joses Workshop*, 2001.

[DP2002] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[dS1990] Fabio Q.B. da Silva. Toward a Formal Framework for Evaluation of Operational Semantics. Technical Report ECS-LFCS-90-126, University of Edinburgh, 1990.

[dS1992] Fabio Q.B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. PhD thesis, University of Edinburgh, 1992.

[Fer1995] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 103–115, 18–21 June 1995.

[FHL+2001] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Rheinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, October 2001.

[FS1999] Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2–3):137–161, November 1999.

[FW1999] Christian Ferdinand and Reinhard Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17((2/3)), 1999.

[GAF+1999] Daniela Genius, Uwe Aßmann, Peter Fritzson, Henk Sips, Rob Kurver, Reinhard Wilhelm, Henk Schepers, and Tom Rindborg. Java and CoSy Technology for Embedded Systems: the JOSES Project. In *Proc. of the European Multimedia, Microprocessor Systems, Technologies for Business Processing and Electronic Commerce Conference (EMMSEC'99)*, 1999.

[GJS1996] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Hec1977] Matthew S. Hecht. *Flow Analysis of Computer Programs,*. Elsevier, 1977.

[Hor2000] Nigel Horspool. Implementing Java exceptions efficiently. Talk at the Dagstuhl seminar 00451, Efficient Implementation of Object-Oriented Programming Languages, 2000.

[JC1999] Aonix Joe Colloca. Embedded real time systems market and the impact of java. Talk at the J-Consortium Meeting in Kyoto, 1999.

[JS2001] Thomas Jensen and Fausto Spoto. Class analysis of object-oriented programs through abstract interpretation. In Furio Honsell and Marino Miculan, editors, *Proceedings of the 4th International Conference, FOSSACS 2001*, Lecture Notes in Computer Science, pages 261–275, 2001.

[JW1995] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 393–407, 1995.

[KS1992] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *4th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 125–140, 1992.

[KU1977] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, January 1977.

[Lau2000] Peeter Laud. Representation analysis – an overview. Technical Report 8603, The JOSES Consortium, 2000.

[Lau2001] Peeter Laud. Analysis for object inlining in java. *Proceedings of the Joses Workshop*, 2001.

[Mar1998] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[Mar1999a] Florian Martin. Experimental Comparison of *call string* and *functional* Approaches to Interprocedural Analysis. In Stephan Jaehnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 63–75, 1999.

[Mar1999b] Florian Martin. *Generation of Program Analyzers*. PhD thesis, Universit¨at des Saarlandes, 1999.

[Mey1988] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mey1989] Bertrand Meyer. Static typing for eiffel, 1989.

[NNH1999] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[NvO1998] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe – definitely. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–170. ACM Press, New York, 1998.

[Pet1994] Mikael Pettersson. RML — A new language and implementation for natural semantics. *Lecture Notes in Computer Science*, 844, 1994.

[Pet1999] Mikael Pettersson. Compiling natural semantics. *Lecture Notes in Computer Science*, 1549, 1999.

[Plo1981] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, 1981.

[Pro1999a] Christian W. Probst. Control flow analysis – an overview. Technical Report 8602, The JOSES Consortium, 1999.

[Pro1999b] Christian W. Probst. Static method invocation – an overview. Technical Report 8601, The JOSES Consortium, 1999.

[Pro2001] Christian W. Probst. Flow sensitive call graph construction for java. *Proceedings of the Joses Workshop*, 2001.

[Pro2002] Christian W. Probst. Control flow analysis for libraries. In *Proceedings of the Static Analysis Symposium*, 2002. to appear.

[RBLP2000] Tobias Ritzau, Marcel Beemster, Florian Liekweg, and Christian W. Probst. Joc - the joses compiler. *The Embedded Systems Show*, 2000.

[RMR2001] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. pages 43–55, 2001. Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA).

[SFA2000] Zhendong Su, Manuel F¨ahndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL)*, pages 81–95. ACM, 2000.

[Shi1991] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[SHR+2000] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, October 2000.

[SRW1999] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape-analysis via 3-valued logic. In *Proc. of 26th ACM SIGACT-SIGPLAN Symposium on Priciples of Programming Languages*, San Antonio, Texas, 1999.

[ST2000] P. F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *Proceedings of the 8th International Symposium on the Foundations of Software Engineering*, 2000.

[TP2000] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Lecture Notes in Computer Science, 2000.

[WM1995] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.