A Control-Flow-Analysis for Multi-Threaded Java with Security Applications

Diplomarbeit

von

Jörg Bauer

FR 6.2 Informatik
Universität des Saarlandes
Im Stadtwald – Bau 45
Postfach 15 11 50
66041 Saarbrücken

Saarbrücken, 05.03.2001

Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 05.03.2001

Jörg Bauer

Abstract

This work is mainly about four topics: formal semantics of Java, control flow analysis, concurrency in Java and security. In an introduction, a brief overview over all these topics shall be given – employing some examples. We will present several approaches from literature to formalize Java semantics. In our case, we will exclusively consider structural operational semantics as introduced by Plotkin [13]. After that, control flow analysis in the style of [12] will be introduced by means of a small functional calculus. Finally, we shall make the reader familiar with the notion of security models, in particular we shall define the Bell-LaPadula security model [1]

The second section deals with formally defining a (almost proper) subset of Java. This means defining a structural, operational semantics for the chosen language. The presented language, Tiny Java or TJ, incorporates the most of the sequential part of Java as well as class declarations and inheritance, in particular hiding fields and overwriting methods are supported.

In the third part of this work, we shall introduce a control flow analysis for TJ, which determines which kinds of objects and references to them there can appear at certain program points. Properties such as existence of solutions, computability of solutions via a constraint formulation and semantic correctness will be proven. Additionally, an implementation is provided, that actually performs this analysis on small examples. It is written in Standard ML and makes use of fixpoint computation.

Section 4 is all about the only of the four main topics not covered yet: concurrency in Java. We will present a way of formally describing the Java concurreny model, which consists of so-called threads, in terms of event spaces. This notion was introduced by [3]. We will extend TJ and its semantics to include these additional constructs, which rather constrain the execution of multi-threaded programs than providing some means of computation.

In section 5, we will demonstrate how the analysis can be applied to both single-threaded and multi-threaded programs. It is used to safely predict run-time errors, that are mostly type-related, which is due to the design of the analysis. The analysis is actually performed on two small examples with the help of the given implementation. As with the multi-threaded case, we bridge the gap between semantics and concurreny on the one hand and security aspects on the other. In fact, the execution of multi-threaded programs is embedded into a Bell-LaPadula security model, thus provably safe statements about security can be made.

The work concludes with some remarks and comments about alternative approaches, further explanations, future goals and the contribution of this work. Additionally, the complete sources of the implementation are provided.

Zusammenfassung

Diese Arbeit berührt im wesentlichen die folgenden vier Gebieten: formale Java Semantiken, Kontrollflussanalyse, Nebenläufigkeit in Java und Sicherheit. Anhand von Beispielen werden all diese Gebiete kurz eingeführt, wobei der Semantikteil mehrere bekannte Ansätze aus der Literatur vorstellt, die alle auf strukturellen, operationalen Semantiken, wie von Plotkin beschrieben [13], basieren. Anschliessend erfolgt eine Beschreibung von Kontrollflussanalyse à la [12] mit Hilfe einer kleinen funktionalen Sprache, bevor die Einleitung mit einem Abschnitt über Sicherheitsmodelle, insbesondere das von Bell und LaPadula [1], abschließt.

Im zweiten Kapitel wird eine Teilmenge von Java formal definiert, was insbesondere die Definition einer strukturellen, operationalen Semantik beinhaltet. Die betrachtete Teilsprache, Tiny Java oder TJ, umfasst den größten Teil des sequentiellen Anteils von Java sowie Klassendeklarationen und Vererbung, insbesondere das Überschreiben von Methoden und das Verdecken von Feldern.

Teil 3 dieser Arbeit führt eine Kontrollflussanalyse für die Sprache TJ ein. Diese Analyse bestimmt, welche Art von Objekten und Referenzen auf diese Objekte an welchen Programmpunkten auftauchen können. Von dieser Analyse werden Eigenschaften wie Existenz von Lösungen, Berechenbarkeit von Lösungen anhand einer Formulierung durch Constraints und nicht zuletzt semantische Korrektheit bewiesen. Zusätzlich wurde eine Implementation der Analyse in Standard ML in Form einer Fixpunktberechnung angefertigt.

Im vierten Teil geht es um das noch unbetrachtete der eingangs erwähnten vier Themen dieser Arbeit, nämlich um Nebenläufigkeit in Java. Es wird eine auf [3] basierende Möglichkeit angegeben, um nebenläufige Java Programme, d.h. Programme mit mehreren Threads, zu beschreiben: Event Spaces. Diese stellen Bedingungen für die Ausführbarkeit operationaler Regeln bereit, ohne jedoch selbst die Form operationaler Regeln zu haben. Mit ihrer Hilfe wird die bisherige Semantik erweitert, um dann auch Nebenläufigkeit formal fassbar zu machen.

Schließlich kommen wir im fünften Abschitt dann zu typischen Anwendungen der Analyse aus Kapitel 3, sowohl im sequentiellen als auch im nebenläufigen Fall. Es wird gezeigt, wie die Analyse dazu beitragen kann, Laufzeitfehler vorherzusagen. Die Form der Analyse führt dazu, dass dies meist typenbezogene Fehler sein werden. Die Analyse wird anhand von zwei Beispielen konkret vollzogen, wobei die vorher erwähnte Implementation verwendet wird. Was den nebenläufigen Fall angeht, so wird hier die Brücke geschlagen zwischen dem Bisherigen und dem Bell-LaPadula Sicherheitsmodell. Die Ausführung nebenläufiger Programme wird in ein solches Modell eingebettet, was sie zugänglich macht für beweisbar sichere Aussagen

über Sicherheitsaspekte.

Die Arbeit schließt mit einigen Kommentaren und Ausblicken, Alternativen zum dargestellten Ansatz sowie Erläuterungen zu möglicherweise unklar gebliebenen Punkten. Außerdem wird der Beitrag dieser Arbeit zur Forschung erläutert. Im Anhang wird letztlich noch der gesamte Quellcode der Implementation zur Verfügung gestellt.

Acknowledgements

First of all, I want to thank Hanne Riis Nielson, who had the initial idea of designing a control flow analysis for a concurrent Java subset and combining it with security. Furthermore, she provided guidance and advice along the way. Further thanks go to Reinhard Wilhelm for accepting this work as a diploma thesis at his chair. Special thanks shall also go to my family for all their support during the time I wrote this thesis in particular and throughout all my studies in general.

Contents

1 Introduction

1.1 Formal Semantics of Java

The object-orientated programming language Java becomes more and more popular among programmers and researchers. Still, no formal semantics based on a mathematical definition exists, which is absolutely neccessary for researchers to argue about formal correctness. Basically everyone, who tries to do formal proofs, relies on an own translation of the language specification given in [10] into mathematically defined formal semantics. So do we. In section 2, we will specify a subset of Java, called *TinyJava* - or TJ - and define a Plotkin-style [13] structural operational semantics (SOS) of it. In what follows, we will present a chronological overview over work, that has been done in this area; i.e. we will have a look at some other examples of SOS for Java subsets. Their grammars will be given and the design of their semantics is commented on and compared to ours; certainly everything is related to the various goals of the works. This is meant to make the reader become more familiar with typical interests, methods and notions in this part of research. Eventually, we will investigate how concurrency primitives can be incorporated in the semantics and which additional techniques there are applied when dealing with concurrency in SOS.

A first approach to be presented here, is from Drossopoulou and Eisenbach [6]. They concentrate on developping a type system for a subset of Java, called $Java_s$, which includes some primitive types, classes and inheritance, interfaces, dynamic method binding, object creation with `new`, arrays, shadowing of instance variables plus some exceptions. Their language provides us with - concerning statements - assignments, if-then-else and expression statements along with variables, constant values, object creation and method invocation on the expression side, see figure 1 for an excerpt of the grammar. We have slightly adapted their grammar to make it more similar and more comparable to both the other languages presented here and to TJ. C, m, f and x range over class, method, field identifiers and variables respectively; values can be either arrays, implicit references to objects or values of simple types; $Exprs$ is a comma separated list of expressions; and $Args$ is a list of types and identifiers. A major difference to TJ is, that they allow arrays of simple types (they do not further specify these, e.g. integer, truth values or characters). Additionally, but not shown in figure 1, they require explicit declarations, i.e. signatures of classes consisting of method and field identifiers plus types; declarations of interfaces - which we do not consider at all - and global variables. In our language, field declarations are allowed to be written in the classes themselves - like in Java. Plus, we allow local variables, which makes the semantics more complicated. So apart from interfaces, arrays and only global variables TJ, is pretty similar to Drossopoulou's and Eisenbach's $Java_s$.

Their major goal is to deduce a type safety result, so a large part of the work deals with static semantics. Finally, they can show a subject reduction result, which is a little more complicated due to the introduction of a small number of exceptions, that can occur under evaluation (they include a null pointer exception and three array bound related exceptions). From their static semantics, we have borrowed the notation of Γ for the class environment, which is computed once and then used in various contexts. Rather similar are our domains in the operational semantics. They model the state as a pair of a stack and a heap component, like

$$State \;=\; (Id \;\rightarrow\; Value) \times \;(Ref \;\rightarrow\; ObjectOrArray)$$

where identifiers are mapped to values and references point to both objects and arrays in the heap. Also the way they model objects, basically as a record of field names and values annotated with a class name to handle dynamic method invocation, had some influence on the TJ SOS.

$$
\begin{array}{lcl}
Program & ::= & ClassDecls \\
ClassDecls & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ Meths\ \} \\
 & & \quad ClassDecls \ \mid\ \epsilon \\
Meths & ::= & m(Args)\ \{\ Stms;\ \texttt{return}\ Expr\ \} \\
 & & \quad Meths \ \mid\ \epsilon \\
Stms & ::= & \texttt{skip} \ \mid\ Stms;\ Stms \\
 & \mid & \texttt{if}\ Expr\ \texttt{then}\ Stms\ \texttt{else}\ Stms \\
 & \mid & Var\ :=\ Expr\ \mid\ Expr \\
Expr & := & Value\ \mid\ Var \\
 & \mid & Expr.m(Exprs)\ \mid\ \texttt{new}\ C \\
 & \mid & \texttt{new}\ SimpleType([Expr]^{+}) \\
Var & ::= & x\ \mid\ Var.f \\
 & \mid & \texttt{this}\ \mid\ Var[Expr]
\end{array}
$$

Figure 1: $Java_s$ by Drossopoulou and Eisenbach

The next work, which we will look at, is [7] by Flatt, Krishnamurthi and Felleisen. As the title suggests, *Classes and Mixins* is (almost) all about, how the Java class system could be extended with mixins, i.e. class-to-class functions, in the words of the authors. There is some amount of work on static and structural operational semantics for the common Java part of their *ClassicJava*, though. We will concentrate on this part and on their SOS of it, which is to be described now. For a brief overview over *ClassicJava*, consider figure 2, where $Exprs$, $Args$, m, f, x and C are like above and I ranges over interface names. The Kleene star has its common meaning and is used for abbreviation. Like in $Java_s$, interfaces are included. Apart from that, they consider methods to be exactly one expression, statements are completely omitted – they are mainly interested in analyzing the problems based on class and interface hierarchies in combination with mixins. They lack primitive types, too. In addition to $Java_s$ and TJ, Flatt, Krishnamurthi and Felleisen consider method invocation through `super`. Local variable declarations are only possible in a `let` construct, which makes life easier in the semantics (see below). Also in TJ, but not in [6], we have casts, which are called `view` here.

Interesting aspects of the SOS of *ClassicJava*: The authors define a number of relations and predicates to get hold of valid programs, e.g. that the class hierarchy is well-formed, particularly it does not contain any cycles. These predicates are highly formalized and are used to make some proper assumptions about types, too. As we are not especially interested in typing properties, we have chosen not to make too explicit statements about well-formedness of TJ programs; it is a possible extension, though. The SOS is given in form of a term rewrite system. In an auxiliary expression, objects are added to the grammar as a primitive. They play the role references play in $Java_s$ and in TJ. As local variables can only be declared in a let-construct, a particular state component mapping identifiers to values becomes redundant. In the body of the let, every occurence of the declared variable is merely replaced in β-reduction manner by the assinged expression. The store, as they call it, looks like this

$$Store \ = \ Object \ \to \ (ClassId \ \times \ (FieldId \ \to \ Value))$$

The correctness result states, that well-typed programs of type t do not get stuck; in the sense that either they terminate yielding a value of a subtype of t or one of two errors occurs, both of which are part of the semantics. However, [7] is mainly about type theory of mixins. What is presented here is just meant to be some basics underneath a higher-order mixin system plus type theory. Eventually, a soundness result for a mixin-extended version of the language is stated.

The latest paper to be considered here, [11] by Igarashi, Pierce and Wadler,

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{ClassDecls}^* \ \mathit{Expr} \\
\mathit{ClassDecls} & ::= & \texttt{class}\ C\ \texttt{extends}\ C' \\
 & & \quad \texttt{implements}\ I^*\ \{\ \mathit{Fields}\ \mathit{Meths}\ \} \\
 & | & \texttt{interface}\ I\ \texttt{extends}\ I^*\ \{\ \mathit{Meths}\ \} \\
\mathit{Fields} & ::= & \mathit{Type}\ f \quad \mathit{Fields}\ \mid\ \epsilon \\
\mathit{Meths} & ::= & \mathit{Type}\ m(\mathit{Args})\ \{\ \mathit{Body}\ \} \quad \mathit{Meths}\ \mid\ \epsilon \\
\mathit{Body} & ::= & \mathit{Expr}\ \mid\ \texttt{abstract} \\
\mathit{Expr} & ::= & \texttt{new}\ C\ \mid\ \mathit{Var}\ \mid\ \texttt{null} \\
 & | & \mathit{Expr}.m(\mathit{Exprs})\ \mid\ \texttt{super}.m(\mathit{Exprs}) \\
 & | & \texttt{view}\ \mathit{Type}\ \mathit{Expr}\ \mid\ \texttt{let}\ \mathit{Var}\ =\ \mathit{Expr}\ \texttt{in}\ \mathit{Expr} \\
\mathit{Type} & ::= & C\ \mid\ I \\
\mathit{Var} & ::= & x\ \mid\ \texttt{this}
\end{array}
$$

Figure 2: *ClassicJava* by Flatt, Krishnamurthi and Felleisen

deals with the problem of finding a core calculus for Java, similar to what the λ-calculus means to functional programming languages. It is no surprise, that *Featherweight Java*, FJ (figure 3), differs most from what we have seen so far, mainly in size. Still, it demonstrates some techniques, which have also been used in the design of TJ, and which apply to most object orientated calculi/languages. Notation, i.e. metavariable conventions in particular, is as usual. Only once, we had to deviate from our common notation, namely at the same place where FJ deviates from both *ClassicJava*, $Java_s$ and TJ: there are constructors in FJ. The notation means, that for every class there is a unique constructor, which, at object creation, assigns some initial values to the fields of the freshly created object. In all the other languages presented and to be presented, `new` is not given any arguments, but initializes fields with intitial values like 0 for integers, `false` for truth values, `null` for any object and so on. Once again, we do not have any means of computation except for some few expressions. Primitive types and values are missing, too.

The SOS is rather simple, because it only has three rules. Objects are the only values and no state is needed, except for a statically determined class table (our notation CT for a list of class declarations has its origin here, see section 2) to support method and field lookup - the equivalent of Γ from before. Still, they can show some soundness results, which can be proven quite easily due to the size of the language. On top of that, they introduce generic classes to FJ and prove soundess results for the extended language, too. This is one of the major benefits of such a small calculus. It can be easily extended and already proven results can be reused as a basis for results about the extended language. Indeed, we originally intended to analyze an FJ-like language instead of TJ (see final section for a discussion).

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \mathit{ClassDecls}\ \mathit{Expr} \\
\mathit{ClassDecls} & ::= & \texttt{class}\ C\ \texttt{extends}\ C' \\
 & & \quad \{\ \mathit{Fields}\ \mathit{Con}\ \mathit{Meths}\ \}\quad \mathit{ClassDecls}\ \mid\ \epsilon \\
\mathit{Fields} & ::= & C\ f\ \mathit{Fields}\ \mid\ \epsilon \\
\mathit{Con} & ::= & C(C^*\ f^*)\ \{\texttt{super}(f^*);\ \texttt{this}.f^*\ =\ f^*;\} \\
\mathit{Meths} & ::= & C\ m(\mathit{Args})\ \{\ \texttt{return}\ \mathit{Expr};\ \} \\
\mathit{Expr} & ::= & x\ \mid\ \mathit{Expr}.f\ \mid\ \mathit{Expr}.m(\mathit{Exprs}) \\
 & \mid & \texttt{new}\ C(\mathit{Exprs})\ \mid\ (C)\mathit{Expr}
\end{array}
$$

Figure 3: FJ by Igarashi, Pierce and Wadler

As we eventually aim at analyzing multi-threaded Java programs, too, two papers by Cenciarelli, Knapp, Reus and Wirsing [3, 4] must be mentioned. They, just like we will do, do not bother with the soundness of a type system, but try to give a SOS for a part of Java close to ours in a multi-threaded setting. To cope with this, they make use of so called event spaces. Further information about these structures and [3, 4] will be given in section 4.

1.2 Control Flow Analysis

The particular form of control flow analysis(CFA), which is applied here, is derived from [12, chapter 3]. In this work by F.Nielson, H.R.Nielson and Hankin, a CFA for a small functional language is developped. We will briefly present some key concepts in this introduction, which will be important for the intuition behind the analysis described in section 3. We will not mention all the required formalisms behind this analysis, but merely conduct ideas. In general, CFA aims at statically determining, how control flows in a given program. This is nontrivial for functional and object-orientated languages, where we are confronted with the *dynamic dispatch problem.*

Consider a small functional language like this

$$
\begin{array}{lcl}
e & ::= & t^l \\
t & ::= & c\ \mid\ x\ \mid\ \texttt{fn}\ x\ \Rightarrow\ e_0\ \mid\ e_1\ e_2 \\
 & \mid & \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \mid\ \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2\ \mid\ e_1\ op\ e_2
\end{array}
$$

where we have constants, variables, λ-abstraction, recursion, if-then-else, local variable binding and binary operations in ML-like syntax. The first interesting aspect of this approach to CFA is the labelling of terms. Expressions are just labelled terms in this language. The labels, imagine e.g. natural numbers, are used to describe, *where* in the program the control is – at a certain label.

How can an analysis be designed, and what exactly does it determine? It determines for each label a set of *abstract values*. These abstract values must be a description of any possible concrete value (in the semantics), that can occur at a particular label at run-time. In the case of the functional language defined above, abstract values are sets of terms. Variables are also mapped to abstract values: for each variable, the associated abstract value must be a description of whatever concrete values the variable can assume under execution of the program. To sum it up, we have the following abstract domains:

$$\begin{array}{lcl} \widehat{\mathbf{Value}} & = & \mathcal{P}(\mathbf{Term}) \\ \widehat{\mathbf{Env}} & = & \mathbf{Var} \rightarrow \widehat{\mathbf{Value}} \\ \widehat{\mathbf{Cache}} & = & \mathbf{Label} \rightarrow \widehat{\mathbf{Value}} \end{array}$$

So an analysis result is a pair of an abstract cache and an abstract environment. The specification of the analysis is given in form of an *acceptability relation*

$$\models \subseteq (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{Exp}})$$

This relation says, whether a pair of cache and environment is an acceptable analysis of an expression. We will also use such a relation for the CFA of TJ. The domains are slightly more complicated, though. What is left to do then, is for each term and an analysis result, to say whether it is acceptable or not. One easy example of this is given by:

$$(\hat{C}, \hat{\rho}) \ \models \ x^l \ \mathtt{iff} \ \hat{\rho}(x) \subseteq \hat{C}(l)$$

This condition on the acceptability of an analysis result $(\hat{C}, \hat{\rho})$ illustrates the interplay between cache and environment. A result for x^l is acceptable if and only if every possible value for x is included in the possible values at label l. Information flows from x to l. Certainly, not all the cases are so easy, but it gives a rough idea about how this specification works.

Now, we have an analysis result and know, whether it is acceptable. But how have we obtained this result? Guessing? Possible, but not always feasible, so we have to think of something else. The solution is closely related to the issue of well-definedness of the relation. Given a specification of an acceptabtibility relation, does it really define a relation. Does there always exist a solution? These questions are answered by using lattice theory. One has to view the clauses of a specification as defining a function

$$(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{Exp}}) \;\rightarrow\; (\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{Exp}})$$

If all our domains are finite and define complete lattices and our clauses define a montone function, then lattice theory tells us, roughly, that a solution exists and can be obtained by fixpoint iteration. This approach will be used for our analysis, too. In reality, it is not as simple as it might seem, but as we said: get an idea. For full details and a discussion of all the problems, that can arise doing this kind of analysis for this particular functional language, consider [12, chapter 3].

Eventually, we want to point to correctness. What does correctness actually mean in the context of a control flow analysis? To answer this, we have to define a formal semantics of the language of interest. If so, we can obtain subject reduction results, similar to those in type theory. Actually, the term is borrowed from type theory. To do so, we have to define correctness relations between the concrete domains of a semantics and the abstract domains of a CFA. These relations say something about, whether an abstract domain, say an abstract state, is a good description of a concrete one. Then we have to prove, that these relations are maintained under evaluation. Additionally, we have to show, that an acceptable analysis result for an expression also is an acceptable result for the evaluated expression (by the semantics). All this will be done in section 3 for TJ and has been done for several other languages like the one above [12, 8, 2].

1.3 The Bell-LaPadula Security Model

In this part of the introduction, we shall present the so-called Bell-LaPadula security model (BLP) invented by Bell and LaPadula [1]. The original paper is quite long, so we recommend [9, chapter 4] for a good introduction. Our introduction will be mainly based on this work, too. For further details on security models and related topics consult [9] as well. We will restrict ourselves to presenting the essentials, that we need to specify properties of the forthcoming analysis in section 5.2.

We start our description of BLP from the following sets

- a set of subjects S. These are the active parts in a system. They can either *observe* or *alter* the passive parts of the system namely

- a set of Objects O that are observed or altered by subjects
- a set of access operations $A = \{execute, read, append, write\}$. The names of the access operations are historic and root in the original BLP, which was designed to state security policies of multi-user operating systems. But as the authors themselves say in [1], their model is not restricted to operating systems. Execute, read, append and write represent the four possibilities of a subject's access to an object: A subject might neither observe nor alter an object (execute), it might observe without altering (read), or vice versa (append) or even alter and observe at the same time (write).
- a set L of security levels, where L is supposed to be a partially ordered set with order $\leq$

We want to use the state of a system – in our context, it will be the state of a computation in terms of a semantic configuration – for checking its security, so the state set of our model has to capture all current permissions and all current instances of subjects accessing objects. This leads to a state set $B \times \mathcal{M} \times F$, where

- $B = \mathcal{P}(S \times O \times A)$ is the set of current accesses. An element $b \in B$ is a collection of tuples (s, o, a), indicating that subject s currently performs an a access on object o.
- $\mathcal{M}$ is the set of access permission matrices $M = (M_{so})_{s \in S, o \in O}$. For each subject s and object o an access permission matrix M has an entry at (s, o), that is the set of access operations, that s is allowed to perform on o.
- $F \subseteq (S \to L) \times (S \to L) \times (O \to L)$ is the set of security level assignments. An element $f \in F$ is a triple (f_S, f_C, f_O) where
  - f_S gives the maximal security level each subject can have
  - f_C gives the current security level of each subject. Of course it cannot be higher than the maximal level, hence $f_C(o) \leq f_S(o)$ for all $o \in O$.
  - f_O gives the classification of all objects, another term for the security level of the objects

BLP defines security as a property of states (in our case one might think of semantic configurations). A state is called *secure* if it satisfies the following three properties

ss-property A state (b, M, f) satisfies the ss-property, if for each element $(s, o, a) \in b$ where the access operation a is read or write, the security level of the subject s dominates the classification of the object o, i.e. $f_O(o) \leq f_S(s)$. This is also called *no read-up.*

$\star$-property A state (b, M, f) satisfies the $\star$-property, if for each element $(s, o, a) \in b$ where the access operation a is append or write, the current level of the subject s is dominated by the classification of the object o, i.e. $f_C(s) \leq f_O(o)$. This is also called *no write-down.* Furthermore, if there exists an element $(s, o, a) \in b$ where the access operation a is append or write, then we must have $f_O(o') \leq f_O(o)$ for all objects o' with $(s, o', a') \in b$ and a' is read or write.

ds-property A state (b, M, f) satisfies the ds-property, if for each element $(s, o, a) \in b$ we have $a \in M_{so}$

A system is called *secure* if all states that it might assume are secure. We will use this notion of a secure system and transfer it to secure programs later on in section 5.2. For a discussion about these properties, their names and why they are chosen to be just like that, consult [9] or [1]. We merely wanted to introduce this model as one of the most famous security models in literature, because we will need it later on and impose it on our semantics. Motivation behind it, as well as features and flaws of it, shall not be of interest in this work.

2 The Tiny Java Language

We will now introduce our language of interest: Tiny Java or TJ. Later on, we will specify a control flow analysis of it, and finally it will be extended with concurrency primitives – just like *ClassicJava* had been extended with mixins and FJ with generic classes. The reader should be familiar with object-orientated concepts like inheritance as well as with Java. We will not start from the scratch. In the first part, we will give the abstract syntax of TJ and in the second, a structural operational semantics will be developped.

2.1 Syntax

A TJ program consists of a list of class declarations and one statement. This is meant to be the Java main method. Like the original, it does not return any value, so it is just a statement and not an expression. Classes are declared by giving their predeccessor class or superclass, i.e. the class, which they inherit from, and a list of field and method declarations. We have not included constructors; fields will be initialized by some initial values like 0 or `false`. Both fields and methods are equipped with an access modifier, which can either be public or private. The meaning is like in Java, i.e. a call to a private method may only occur in the defining class itself – *syntactically.* The same applies to field access. Because of simplicity, we have dropped protected modifiers and the concept of packages. We provide if-then-else, assignments, local variable declarations, sequencing and expression statements. While-statements have been dropped, because their computational power can be encoded in recursive methods and they do not really add something interesting to the semantics and to the correctness proof in section 3. Local declarations, however, add an interesting feature to a SOS, just like method calls do, and are actually dealt with in a similar way. Of course, there are constants of some primitive types (bool and integer), some operations, variables, object creation and method invocation among the expressions. Quite interesting here, especially with respect to the analysis and its results, are casts and instanceof expressions. Note, that we require `this` always to be written, so method calls and field access are always of the form $x.f$ or $x.m$. The grammar defining TJ is given in figure 4. On top of the grammar rules, we agree on the following: A valid TJ method has exactly one return statement. This statement is the last one in a method. This is actually not a major restriction, because every valid Java method can be easily transformed into one with exactly one return in the end. Furthermore, we require `this` not to be written in the main method. *binop* includes a collection of binary arithmetic and logic operators as well as relations. *binop* and *unop* –

$$
\begin{array}{lcl}
\mathit{Program} & ::= & \mathit{ClassDecls}\ \mathit{Stms} \\
\mathit{ClassDecls} & ::= & \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \mathit{Fields}\ \mathit{Meths}\ \} \\
 & & \quad \mathit{ClassDecls}\ \mid\ \varepsilon \\
\mathit{Fields} & ::= & \mathit{AccMod}\ \mathit{Type}\ f;\ \mathit{Fields}\ \mid\ \varepsilon \\
\mathit{Meths} & ::= & \mathit{AccMod}\ \mathit{Type}\ m(\mathit{Args})\ \{\ \mathit{Stms}\ \} \\
 & & \quad \mathit{Meths}\ \mid\ \varepsilon \\
\mathit{LocalDecls} & ::= & \mathit{Type}\ x\ =\ \mathit{Expr};\ \mathit{LocalDecls}\ \mid\ \varepsilon \\
\mathit{Args} & ::= & \mathit{Type}\ x, \mathit{Args}\ \mid\ \mathit{Type}\ x\ \mid\ \varepsilon \\
\mathit{AccMod} & ::= & \texttt{private}\ \mid\ \texttt{public} \\
\mathit{Type} & ::= & \texttt{void}\ \mid\ \texttt{bool}\ \mid\ \texttt{int}\ \mid\ C \\
\mathit{Stms} & ::= & \texttt{skip}\ \mid\ \mathit{Expr}\ \mid\ \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stms}\ \texttt{else}\ \mathit{Stms} \\
 & \mid & \mathit{Stms};\ \mathit{Stms}\ \mid\ \{\ \mathit{LocalDecls}\ \mathit{Stms}\ \}\ \mid\ \mathit{Var}\ :=\ \mathit{Expr} \\
 & \mid & \texttt{return;}\ \mid\ \texttt{return}\ \mathit{Expr}; \\
\mathit{Expr} & ::= & \texttt{n}\ \mid\ \texttt{true}\ \mid\ \texttt{false}\ \mid\ \texttt{null} \\
 & \mid & \mathit{Expr}\ \mathit{binop}\ \mathit{Expr}\ \mid\ \mathit{unop}\ \mathit{Expr} \\
 & \mid & (C)\mathit{Expr}\ \mid\ \mathit{Expr}\ \texttt{instanceof}\ C \\
 & \mid & \mathit{Var}\ \mid\ \texttt{new}\ C() \\
 & \mid & \mathit{Expr}.m(\mathit{Exprs}) \\
\mathit{Exprs} & ::= & \mathit{Exprs},\ \mathit{Expr}\ \mid\ \mathit{Expr}\ \mid\ \varepsilon \\
\mathit{Var} & ::= & x\ \mid\ \texttt{this}\ \mid\ \mathit{Var}.f
\end{array}
$$

Figure 4: Tiny Java Abstract Syntax

also a collection of common unary operators – are left unspecified, because they do not really influence the analysis to be considered. Words written in `typewriter` are terminals of the grammar, words in *italics* nonterminals. The metavariables C, x, f and m range over distinct sets of class, variable, field and method names, res. ε stands for the empty word. `Object` must not be in the set of class names occuring in the program and `this` must not be used as a variable name. n is supposed to be a literal consisting of digits, possibly with a leading minus, and represents integers.

2.2 Semantics

The operational semantics of TJ will be given in Plotkin-style SOS [13]. We shall first define the semantic domains and give an example to illustrate these. Then we have to make some assumptions about the static semantics, which is not explicitely given here (Good descriptions of static semantics of TJ-like languages can be found in the papers presented in section 1.1 [6, 7, 11]), before we come to the actual semantic rules. This section concludes with a sample run of the example. But let us start with some definitions and conventions.

Definition 2.1 (Semantic Domains) *Let P be a program according to the syntax defined above. We define the following semantic domains together with metavariables ranging over them.*

$$\begin{array}{rcll}
C & \in & \mathbf{ClassId} & \textit{the set of class names in } P. \\
m & \in & \mathbf{MethId} & \textit{the set of method names in } P. \\
f & \in & \mathbf{FieldId} & \textit{the set of field names occuring in } P. \\
x & \in & \mathbf{Id} & \textit{the set of variable names occuring in } P. \\
r & \in & \mathbf{Adr} & \textit{a not furtherly specified, countably infinite} \\
 & & & \textit{set of adresses or locations in the heap.} \\
acm & \in & \mathbf{AccMod} & := \mathbf{ClassId} \cup \{\triangle\} \\
r_C & \in & \mathbf{Ref} & := \mathbf{Adr} \times \mathbf{ClassId} \\
z & \in & \mathbf{Z} & \textit{integers} \\
b & \in & \mathbf{B} & := \{\top, \bot\} \textit{ truth values} \\
v & \in & \mathbf{Value} & := \mathbf{Z} \cup \mathbf{B} \cup \mathbf{Ref} \cup \{\diamond, \star\} \\
o, \varphi_C & \in & \mathbf{Object} & := \mathbf{ClassId} \times ((\mathbf{FieldId} \times \mathbf{ClassId}) \to \mathbf{Value}) \\
\langle \sigma, \kappa \rangle & \in & \mathbf{State} & := ((\mathbf{Id} \cup \{\mathtt{this}\}) \to \mathbf{Value}) \;\times\; (\mathbf{Adr} \to \mathbf{Object})
\end{array}$$

With all metavariables ranging over a semantic domain, so do the metavariables annotated with primes or indices.

Adr must be countably infinite, because we dynamically create new references in the semantics without checking boundaries. An access modifier being a class name denotes, that a field or method is private to that particular class, whereas $\triangle$ means public. $\top$ means true, $\perp$ false; $\diamond$ stands for a null reference and $\star$ is *no value* used for returning something from a method declared `void`.

In order to explain the design of the other domains, we have to say something about important object-orientated concepts in Java and TJ: method overriding resulting in dynamic method binding, and shadowing of fields. These concepts can be best explained with the help of an example. Consider the following TJ program (we have added Java-like comments, which are not included in the syntax):

```
class A extends Object {
  public int a;
  public void m() { this.a = 1; return;}
}

class B extends A {
  public int a;
  public void m() { this.a = 2; return;}
}

{
  A a = new A();
  A ab = new B();
  B b = new B();
  a.m();                    // a=1
  b.m();                    // a=2
  ab.m();                   // ab.a=0
  ab.a == 2;                // false !!
  (B(ab)).a == 2;           // true !!
  b.a = 1;
  ((A)b).a = 2;
  ((B)b).a == 1;            // true !!
}
```

This little program demonstrates important object-orientated concepts. It shows the difference between reference types (the class name references are annotated with) and object types (the class names objects are annotated with). We create three objects, two of which have type B and one type A.

The type of the object is determined at creation! This type is part of the semantic domain of an object. Furthermore, an object contains fields, which are mapped to values in the semantic domain. Methods are not part of an object but part of a class. An object contains all the fields declared in its class type plus all the fields declared in superclasses. These names are not neccessarily distinct, like in our example. An object of type B has two fields a with the same name. This is called *shadowing*. Both fields remain accessible, though. In our example, a of class A is shadowed in class B. Methods can be *overwritten*. This means, that a class extending another class can redefine a method, which already existed in the other class. The redefined method must have the same signature, though. Otherwise, this is called *overloading*, which we will not consider here. In our example, class B overwrites the method m of class A. How do we know which field and which method to access? The most important fact about reference and object types is: The object type determines which method, the reference type determines which field is accessed. In our example: When `a.m()` is called, a is an A reference to an A object, thus the method m in A is called, and the only field of an A object is assigned 1. The same happens with the next method call. A B reference to a B object certainly calls the method defined in B. So, which field is assigned? Also the one declared in class B. Note, that B objects also have another field, the one inherited. This field is not assigned yet. The most interesting case is ab, an A reference to a B object. If we call m on ab, the m method from class B is selected, thus the field associated with class B is assigned. This field cannot be seen from ab, because it is an A reference and the reference type determines, which fields are accessed. Therefore, the first comparison fails (`ab.a == 2;`). If we cast it down to a B reference, though, the previously assigned field suddenly appears. The last three lines of the sample program, show the same effect. Once we assign the field associated with class A to our B object and once the field associated with B. The last line proves, that both fields are kept in the object.

This typical behaviour is modelled in our semantic domains. A reference is annotated with the reference type, which determines, which fields to access. If we call a method through a reference, we have to look up, which kind of object the reference points to, and then look up the method of the object type. Shadowing is implemented by mapping a field and a class name to a value in the object domain. After having introduced the semantics of TJ, we will perform a sample run on this example, to see that the semantics really performs like Java (at least on this example). Other concepts known from Java like method overloading or static and final fields and methods are not considered in this work and are object to a possible extension.

In order to specify the semantic rules, we shall agree on some more metavariable conventions. We need metavariables ranging over syntactic domains in addition to those ranging over semantic domains as defined in definition 2.1. Note, that some domains are both syntactic and semantic, in particular class, field, method and variable names. We will not distinguish between syntax and semantics there. In what follows let (CT, s) range over programs, CT over class declarations, M over method declarations, F over field declarations, t over types, e over expressions, s over statements, E over $Exprs$, A over $Args$, *decl* over one local declaration, D over $decl^\star$, *uop* over unary operators, *bop* over binary operators and *var* over Var. Note, that ranging over the left-hand side of a production, means ranging over everything, that can be produced by the TJ grammar starting with this left-hand side nonterminal, e.g. *var* ranges over $\{x, \mathtt{this}, x.f, \mathtt{this}.f, x.f_1.f_2, \ldots\}$. Furthermore, we need terms to talk about static well-formedness of a program.

Definition 2.2 (Class Environment) *Let (CT,s) be a program. The class environment* Γ *of (CT,s) is defined as follows:*

$$\Gamma \;:\; \mathbf{ClassId} \;\rightarrow\; \begin{array}{l} (\mathbf{ClassId} \cup \{\mathit{\texttt{Object}}\}\;)\times \\ (\;\mathbf{FieldId} \rightarrow (\mathbf{AccMod} \times \mathbf{Type})\;)\times \\ (\;\mathbf{MethId} \rightarrow (\mathbf{AccMod} \times Type \times Args \times Stms \times Exp)\;) \end{array}$$

is the function, which maps a class name to the triple consisting of its superclass, a mapping from its field names to their respective access modifiers and types, and a mapping from its method names to their signature, consisting of access modifier, return type, arguments, body (i.e. a statement) and return expression. Define $\mathbf{super}_\Gamma$*,* $\mathbf{fields}_\Gamma$ *and* $\mathbf{meths}_\Gamma$ *to be the functions mapping a class name* C *to the first, second and third component of* $\Gamma(C)$ *respectively.*

Γ is merely a transformation of a program into another form and should not need to be further specified. It can be computed by one recursive traversal of the parse tree, meaning that Γ will be computed once and for all. In the course of this section, we will assume Γ to be implicit in every context, when we talk about a program, e.g. it will not be given as an argument to functions that actually use it. Γ derived from the example program given above looks like this:

$$\begin{array}{ll} \Big[& A \mapsto \Big(Object, [a \mapsto (\triangle, \mathtt{int})], [m \mapsto (\triangle, \mathtt{void}, \varepsilon, \mathtt{this.a=1;}, \star)]\Big), \\ & B \mapsto \Big(A, [a \mapsto (\triangle, \mathtt{int})], [m \mapsto (\triangle, \mathtt{void}, \varepsilon, \mathtt{this.a=2;}, \star)]\Big) \;\Big] \end{array}$$

where finite mappings are represented as lists. Furthermore, we write ε, for no arguments and $\star$ for a method declared `void`. Finally, we do not write skip statements.

Definition 2.3 ($\preceq$) *The relation* $\preceq\ \subseteq$ **ClassId** $\times$ **ClassId** *is the reflexive, transitive closure of* $\{(C, super_\Gamma(C)) \mid C \in \mathbf{ClassId}\}$.

Static Properties. Some static properties shall be imposed on programs. We require the graph $(\mathbf{ClassId} \cup \{\texttt{Object}\}, \{(C, super_\Gamma(C)) \mid C \in \mathbf{ClassId}\})$, i.e. the graph with the nodes being the class names in the program and edges according to the class extension, to be a tree with root `Object`. In particular, there are no cycles; every class except the root of the hierarchy has exactly one superclass; classes that are extended are defined. Additionally, we require method and field names within one class to be unique, as we do not deal with overloading. Accessibility properties can be completely checked statically [10, chapter 6.6], so no run-time checks on access modifiers have to be performed. We require overwriting methods and fields shadowing other fields to be at least as accessible as those they overwrite or hide, e.g. a public field f in class A must not be hidden by a private field f in B extending A. This requirement is made in Java, too, and is checked statically. Finally, we assume the program to be well-typed in the sense of the papers in the introduction [6, 7, 11]. We will not give any type rules or even proofs linking a type system to our operational semantics. It has been shown, that this can be done and we will not bother with this. So, we do not include run-time type checks in our operational semantics, either. Our analysis will be able to find out about some types as well. We refer to sections 5 and 6.

The Semantics. As we deal with an SOS, representing local states and method calls is not trivial, because we have to remember the original state and forget about the local state after the method has been evaluated. In terms of compilers, we have to represent the call frames on the stack. Each method call allocates a new frame with new instances of local variables. This has to be modelled in the semantics. [3, 4] model this by explicitely encoding a stack into their semantic domains. We will use the approach described in [12, chapter 2.5 and 3] and introduce *intermediate expressions and statements*. These are pieces of syntax, that are not to be written in a source program, but that are generated during evaluation. Here they are:

$$\begin{array}{lcl} \mathit{Stms} & ::= & \ldots \mid \texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{\mathit{LocalDecls}\ \mathit{Stms}\} \\ \mathit{Exp} & ::= & \ldots \mid r_C \mid \top \mid \bot \mid \star \mid \diamond \mid r_C.f \end{array}$$

We allow elements of the semantic domain **Value** to occur in syntax, because we have to evaluate arguments to a method call one by one yielding a mixture of semantic values and syntax. The same happens at field access. The bind-construct includes a semantic state into the syntax. This state will become the local state, on which we will compute. Nested binds represent the different call frames on the stack. Talking about TJ terms now, we mean terms produced by the extended grammar. Now we can define our semantic relation.

Definition 2.4 (Semantic Relation) *A configuration of a TJ program under evaluation is defined as*

$$\mathbf{Config} \;:=\; (Term \times \mathbf{State}) \;\cup\; \mathbf{State}$$

where Term is the set of strings produced by the TJ grammar in figure 4 plus the productions for intermediate expressions.

$$\rightarrow_\Gamma \quad \subseteq \quad \mathbf{Config} \times \mathbf{Config}$$

is called the semantic relation and is defined by the inference rules in tables 1 to 7. $c \rightarrow_\Gamma c'$ is read "c evaluates to c' with respect to Γ in one step".

[id]	$\dfrac{}{(x, \langle\sigma, \kappa\rangle) \;\rightarrow_\Gamma\; (\sigma(x), \langle\sigma, \kappa\rangle)}$	$x \in dom(\sigma)$
[access1]	$\dfrac{(var, \langle\sigma, \kappa\rangle) \;\rightarrow_\Gamma\; (var', \langle\sigma', \kappa'\rangle)}{(var.f, \langle\sigma, \kappa\rangle) \;\rightarrow_\Gamma\; (var'.f, \langle\sigma', \kappa'\rangle)}$	
[access2]	$\dfrac{}{(r_C.f, \langle\sigma, \kappa\rangle) \;\rightarrow_\Gamma\; (\varphi(f, C), \langle\sigma, \kappa\rangle)}$	$r_C \neq \diamond$ $\kappa(r) = \varphi_{C'}$

Table 1: Variables and Field Access

Table 1 gives the inference rules for variables and field access. Note that field lookup makes use of the class according to the type of the reference, not the object. Then, there is no rule for $\diamond$ being dereferenced, which means that the semantics gets stuck. Others suggest to throw an exception here. Make yourself clear, that recursion in the manner of $x.f_1.f_2$ is dealt with correctly and that rule id handles `this` the right way. In order to specify the inference rules of table 2, we have to define yet another term, l-ground, which is borrowed from [6].

Definition 2.5 (l-ground) *An expression e is l-ground if and only if $e \equiv x$ with $x \in$ **Id** or $e \equiv r_C$ with $r_C \in$ **Ref**.*

[skip]	$\dfrac{}{(\texttt{skip}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma, \kappa \rangle}$	
[assign1]	$\dfrac{(var, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (var', \langle \sigma', \kappa' \rangle)}{(var := e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (var' := e, \langle \sigma', \kappa' \rangle)}$	var is not l-ground
[assign2]	$\dfrac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{(var := e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (var := e', \langle \sigma', \kappa' \rangle)}$	var is l-ground
[assign3]	$\dfrac{}{(x := v, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma[x \mapsto v], \kappa \rangle}$	$x \in \mathbf{Id} \cup \{\texttt{this}\}$ $v \in \mathbf{Value} \setminus \{\star\}$
[assign4]	$\dfrac{}{(r_C.f := v, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma, \kappa[r \mapsto o'] \rangle}$	$r_C \in \mathbf{Ref} \quad \kappa(r_C) = \varphi_{C'}$ $v \in \mathbf{Value} \setminus \{\star\}$ $o' = (\varphi[(f, C) \mapsto v])_{C'}$

Table 2: Skip and Assignment

l-groundness prevents terms on the left-hand side of an assignment from being evaluated too far. Left-hand sides are merely evaluated until they match the terms given in rules assign3 and assign4. Note, that we do not need to distinguish between ordinary variables and `this` here, as the this reference is properly included in σ. In case of several field acceses, assign4 applies to the rightmost field access, because l-groundness requires the leftmost fields to be completely dereferenced. Once again, the relation between field access and reference types is displayed in rule assign4. The new value is assigned to (C, f), not to (C', f).

The rules for if-then-else constructs and sequencing are standard and should not need any further explanation. The difference between an expression and an expression statement is made visible by putting a semicolon behind the expression being a statement. Certainly, the value of an expression is thrown away after evaluation of an expression statement.

Besides method calls, the most interesting rules are the ones for local declarations (4), because we use the `bind` ... `in` ...-construct here. If we enter a new local scope denoted $\{\ldots\}$, we have to keep track of two different states. First, we have to remember the current state at entrance; then we have to

[expstm1]	$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{(e;\,, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e';\,, \langle \sigma', \kappa' \rangle)}$$	
[expstm2]	$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{(e;\,, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma', \kappa' \rangle}$$	$v \in \textbf{Value}$
[if]	$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, \langle \sigma, \kappa \rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{if } e' \texttt{ then } s_1 \texttt{ else } s_2, \langle \sigma', \kappa' \rangle)\end{array}}$$	
[iftrue]	$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (\top, \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, \langle \sigma, \kappa \rangle) \\ \quad \rightarrow_\Gamma \ (s_1, \langle \sigma', \kappa' \rangle)\end{array}}$$	
[iffalse]	$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (\bot, \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, \langle \sigma, \kappa \rangle) \\ \quad \rightarrow_\Gamma \ (s_2, \langle \sigma', \kappa' \rangle)\end{array}}$$	
[seq1]	$$\frac{(s_1, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (s_1', \langle \sigma', \kappa' \rangle)}{(s_1;\ s_2, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (s_1';\ s_2, \langle \sigma', \kappa' \rangle)}$$	
[seq2]	$$\frac{(s_1, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma', \kappa' \rangle}{(s_1;\ s_2, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (s_2, \langle \sigma', \kappa' \rangle)}$$	

Table 3: Expression statements, If and Sequence

[nodecl]
$$\frac{}{(\{s\}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (s, \langle \sigma, \kappa \rangle)}$$

[bind]
$$\frac{}{\begin{array}{l}(\{decl\ s\}, \langle \sigma, \kappa \rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{decl\ s\}, \langle \sigma, \kappa \rangle)\end{array}}$$

[local1]
$$\frac{(decl, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (decl', \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(\texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{decl\ s\}, \langle \sigma_0, \kappa_0 \rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{bind}\ \langle \sigma', \kappa' \rangle\ \texttt{in}\ \{decl'\ s\}, \langle \sigma_0, \kappa_0 \rangle)\end{array}}$$

[local2]
$$\frac{(decl, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma', \kappa' \rangle}{\begin{array}{l}(\texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{decl\ s\}, \langle \sigma_0, \kappa_0 \rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{bind}\ \langle \sigma', \kappa' \rangle\ \texttt{in}\ \{s\}, \langle \sigma_0, \kappa_0 \rangle)\end{array}}$$

[local3]
$$\frac{(s, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (s', \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(\texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{s\}, \langle \sigma_0, \kappa_0 \rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{bind}\ \langle \sigma', \kappa' \rangle\ \texttt{in}\ \{s'\}, \langle \sigma_0, \kappa_0 \rangle)\end{array}}$$

[local4]
$$\frac{(s, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma', \kappa' \rangle}{(\texttt{bind}\ \langle \sigma, \kappa \rangle\ \texttt{in}\ \{s\}, \langle \sigma_0, \kappa_0 \rangle) \ \rightarrow_\Gamma \ \langle \sigma_0, \kappa' \rangle}$$

[decl1]
$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{(t\ x = e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (t\ x = e', \langle \sigma', \kappa' \rangle)}$$

[decl2]
$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (v, \langle \sigma', \kappa' \rangle)}{(t\ x = e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma'[x \mapsto v], \kappa' \rangle} \qquad \begin{array}{l} v \in \mathbf{B} \cup \mathbf{Z} \\ t \equiv \texttt{int} \ \vee \ t \equiv \texttt{bool} \end{array}$$

[decl3]
$$\frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_{C'}, \langle \sigma', \kappa' \rangle)}{(C\ x = e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma'[x \mapsto r_C], \kappa' \rangle} \qquad r_{C'} \in \mathbf{Ref}$$

[decls1]
$$\frac{(decl, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (decl', \langle \sigma', \kappa' \rangle)}{(decl\ D, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (decl'\ D, \langle \sigma', \kappa' \rangle)}$$

[decls2]
$$\frac{(decl, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma', \kappa' \rangle}{(decl\ D, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (D, \langle \sigma', \kappa' \rangle)}$$

Table 4: Local Declarations

work with a new state, the local state. The local state is the one within the bind-construct, the outer state is the state, which our evaluation actually is in, typically denoted $\langle \sigma_0, \kappa_0 \rangle$. Rules local1 and local2 build the new local environment by evaluating local declarations. This is done in rules decl1 - decl3 and decls1, decls2. Rule decl3 should incorporate an assignment validity check, i.e. whether it is allowed to declare such a reference to be of the declared type. Once again, this can be done statically. The new bindings are then added to the state encapsulated in the bind-construct. Local3 makes sure, that evaluation of the local statements is performed in the local state. Note that $\langle \sigma_0, \kappa_0 \rangle$ remains unchanged. Local4 defines what happens at the exit of the local scope. The updates on the heap, which have happened, are maintained, whereas we return to the outer stack. In terms of compilers: We pop a frame from the stack like when a method has been executed. This is because Java and TJ allow local declarations everywhere statements can be.

Pay special attention to rule decl3, too. The expression to be assigned yields a reference of type C', whereas x is declared to be of type C. So x is bound to the same adress pointing to the same object, but with a different view on the object.

The only thing worth noting about constants and operations (table 5) are the semantic functions $[\![\cdot]\!]$ mapping syntax to elements of semantic domains. A sequence of digits is mapped to the integer represented by this sequence; syntactic symbols for arithmetic operations are mapped to the functions themselves and so on. This is standard and not further specified here.

Yet two more definitions dealing with method call and lookup:

Definition 2.6 (Method lookup)

$$\begin{aligned} &\mathit{mlookup} \;:\; (\mathbf{ClassId} \times \mathbf{MethId}) \;\rightarrow \\ &\qquad \mathbf{AccMod} \times \mathit{Type} \times \mathit{Args} \times \mathit{Stms} \times \mathit{Expr} \end{aligned}$$

$$\mathit{mlookup}(C, m) \;:=$$

$$\begin{cases} (\mathit{meths}_\Gamma\ C)(m) & \mathit{if}\ m \in \mathit{dom}(\mathit{meths}_\Gamma\ C) \\ \mathit{mlookup}(\mathit{super}_\Gamma\ C, m) & \mathit{if}\ m \notin \mathit{dom}(\mathit{meths}_\Gamma\ C) \wedge\ C \neq \texttt{Object} \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

The interesting thing about method lookup is the class we start searching in. It is determined by the call function, which itself uses *mlookup*, which is to be defined now.

[constn]	$$\frac{}{(n, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; ([\![n]\!], \langle\sigma, \kappa\rangle)}$$	
[constb1]	$$\frac{}{(\texttt{true}, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (\top, \langle\sigma, \kappa\rangle)}$$	
[constb2]	$$\frac{}{(\texttt{false}, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (\bot, \langle\sigma, \kappa\rangle)}$$	
[unop1]	$$\frac{(e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (e', \langle\sigma', \kappa'\rangle)}{(\mathit{uop}\ e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (\mathit{uop}\ e', \langle\sigma', \kappa'\rangle)}$$	
[unop2]	$$\frac{(e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (v, \langle\sigma', \kappa'\rangle)}{(\mathit{uop}\ e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; ([\![\mathit{uop}]\!](v), \langle\sigma', \kappa'\rangle)}$$	$v \in \mathbf{Value}$
[binop1]	$$\frac{(e_1, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (e_1', \langle\sigma', \kappa'\rangle)}{(e_1\ \mathit{bop}\ e_2, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (e_1'\ \mathit{bop}\ e_2, \langle\sigma', \kappa'\rangle)}$$	
[binop2]	$$\frac{(e_1, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (v, \langle\sigma', \kappa'\rangle)}{(e_1\ \mathit{bop}\ e_2, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (v\ \mathit{bop}\ e_2, \langle\sigma', \kappa'\rangle)}$$	$v \in \mathbf{Value}$
[binop3]	$$\frac{(e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (e', \langle\sigma', \kappa'\rangle)}{(v\ \mathit{bop}\ e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (v\ \mathit{bop}\ e', \langle\sigma', \kappa'\rangle)}$$	$v \in \mathbf{Value}$
[binop4]	$$\frac{(e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; (v_2, \langle\sigma', \kappa'\rangle)}{(v_1\ \mathit{bop}\ e, \langle\sigma, \kappa\rangle) \;\to_\Gamma\; ([\![\mathit{bop}]\!](v_1, v_2), \langle\sigma', \kappa'\rangle)}$$	$v_1, v_2 \in \mathbf{Value}$

Table 5: Constants and Operations

[this1]	$$\frac{(e, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (e', \langle\sigma',\kappa'\rangle)}{(e.m(E), \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (e'.m(E), \langle\sigma',\kappa'\rangle)}$$	
[this2]	$$\frac{(e, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (r_C, \langle\sigma',\kappa'\rangle)}{(e.m(E), \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (r_C.m(E), \langle\sigma',\kappa'\rangle)}$$	$r_C \in \textbf{Ref}$
[args1]	$$\frac{(e_i, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (e_i', \langle\sigma',\kappa'\rangle)}{\begin{array}{l}(r_C.m(v_1,\ldots,v_{i-1},e_i\ E), \langle\sigma,\kappa\rangle) \\ \quad \rightarrow_\Gamma \ (r_C.m(v_1,\ldots,v_{i-1},e_i'\ E), \langle\sigma',\kappa'\rangle)\end{array}}$$	$r_C \in \textbf{Ref}$ $v_j \in \textbf{Value}$ for all $1 \leq j < i$
[args2]	$$\frac{(e_i, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (v_i, \langle\sigma',\kappa'\rangle)}{\begin{array}{l}(r_C.m(v_1,\ldots,v_{i-1},e_i\ E), \langle\sigma,\kappa\rangle) \\ \quad \rightarrow_\Gamma \ (r_C.m(v_1,\ldots,v_{i-1},v_i\ E), \langle\sigma',\kappa'\rangle)\end{array}}$$	$r_C \in \textbf{Ref}$ $v_j \in \textbf{Value}$ for all $1 \leq j \leq i$
[call]	$$\frac{}{\begin{array}{l}(r_C.m(v_1,\ldots,v_n), \langle\sigma,\kappa\rangle) \\ \quad \rightarrow_\Gamma \ (call(r, \langle\sigma,\kappa\rangle, m, v_1,\ldots,v_n), \langle\sigma,\kappa\rangle)\end{array}}$$	$r_C \in \textbf{Ref}$ $v_j \in \textbf{Value}$ for all $1 \leq j \leq n$
[ret1]	$$\frac{(e, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (e', \langle\sigma',\kappa'\rangle)}{\begin{array}{l}(\texttt{bind}\ \langle\sigma,\kappa\rangle\ \texttt{in}\ \{\texttt{return}\ e;\}, \langle\sigma_0,\kappa_0\rangle) \\ \quad \rightarrow_\Gamma \ (\texttt{bind}\ \langle\sigma',\kappa'\rangle\ \texttt{in}\ \{\texttt{return}\ e';\}, \langle\sigma_0,\kappa_0\rangle)\end{array}}$$	
[ret2]	$$\frac{(e, \langle\sigma,\kappa\rangle) \ \rightarrow_\Gamma \ (v, \langle\sigma',\kappa'\rangle)}{\begin{array}{l}(\texttt{bind}\ \langle\sigma,\kappa\rangle\ \texttt{in}\ \{\texttt{return}\ e;\}, \langle\sigma_0,\kappa_0\rangle) \\ \quad \rightarrow_\Gamma \ (v, \langle\sigma_0,\kappa'\rangle)\end{array}}$$	$v \in \textbf{Value}$

Table 6: Method Calls

Definition 2.7 (Call Function) *Let* $r \in \mathbf{Ref}$ *be an adress,* $\langle \sigma, \kappa \rangle \in \mathbf{State}$ *be a state. Let* $C = \kappa(r)$ *be the object type of the object located at adress* r *and let* $m \in \mathbf{MethId}$ *be a method name. Let* $(v_1, \ldots, v_n) \in \mathbf{Value}^\star$ *be a list of values. If*

$$(mlookup(C, m) = (acm, t, t_1\ x_1, \ldots, t_n\ x_n, s, e)$$

then the function **call**

$$call \ : \ \mathbf{Adr} \times \mathbf{State} \times \mathbf{MethId} \times \mathbf{Value}^\star \ \to \ Stms$$

is defined as follows

$$\begin{array}{l} call(r, \langle \sigma, \kappa \rangle, m, v_1, \ldots, v_n) \ := \\ \qquad \boldsymbol{\mathit{bind}}\ \langle [\boldsymbol{\mathit{this}} \mapsto r_C][x_i \mapsto v_i]_{i=1}^{n}, \kappa \rangle \ \ \boldsymbol{\mathit{in}}\ \{s;\ \boldsymbol{\mathit{return}}\ e; \} \end{array}$$

This1 and this2 evaluate the calling expression. This should result in a reference. After that, in args1 and args2, the arguments are successively evaluated from left to right. It is essential here, that we allowed semantic values to occur in the syntax as intermediate expressions. When all the expressions have been evaluated, method lookup is done by the *call* function defined in defintion 2.7. This function takes as arguments the adress of the reference, to which the calling expression has been evaluated. With its help, the object type of the associated object is determined. Note that the reference type does not matter for method lookup. From this object type, we start going up the class hierarchy to look for the method. This is done by *mlookup* (definition 2.6). Once we have found the appropriate method, the method body is bound in a bind-construct with the local state only consisting of the formal parameters bound to the current parameters. `This` is bound to the calling reference seen as a reference to an object of the class, in which the method has been found. The bind-construct corresponds exactly to a new call frame on the execution stack. In this context, the method body is evaluated, until a return statement is met. Note, that we have not dealt with return statements yet. It can easily be proven, that whenever such a statement is met, it appears in a bind-construct. Informally speaking: Return statements can only be the last statement in a method, according to our additional agreements on the TJ grammar, hence it can only be evaluated after a method call. But every time a method is called, its body, thus the return statement, will be bound in a bind-construct. Furthermore, due to the definition of Γ and *call* the return statement will either be of the form `return` e; or `return` $\star$; as occurrences of return with no expression are modelled by returning a non-value. That is why we need $\star$ to be a value, too.

$$[\text{new}] \quad \frac{}{(\texttt{new}\ C, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_C, \langle \sigma, \kappa[r \mapsto new(C)] \rangle)} \quad r \text{ fresh}$$

$$[\text{cast1}] \quad \frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{((C)e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ ((C)e', \langle \sigma', \kappa' \rangle)}$$

$$[\text{cast2}] \quad \frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_{C'}, \langle \sigma', \kappa' \rangle)}{((C)e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_C, \langle \sigma', \kappa' \rangle)} \quad \begin{array}{l} r_{C'} \in \textbf{Ref} \\ \kappa'(r) = \varphi_{C''} \\ C'' \preceq C \end{array}$$

$$[\text{inst1}] \quad \frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (e', \langle \sigma', \kappa' \rangle)}{\begin{array}{l}(e \ \texttt{instanceof} \ C, \langle \sigma, \kappa \rangle) \\ \quad \rightarrow_\Gamma \ (e' \ \texttt{instanceof} \ C, \langle \sigma', \kappa' \rangle)\end{array}}$$

$$[\text{inst2}] \quad \frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_C, \langle \sigma', \kappa' \rangle)}{(e \ \texttt{instanceof} \ C, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (\top, \langle \sigma', \kappa' \rangle)} \quad \begin{array}{l} r_{C'} \in \textbf{Ref} \\ \kappa'(r) = \varphi_{C''} \\ C = C'' \end{array}$$

$$[\text{inst3}] \quad \frac{(e, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_C, \langle \sigma', \kappa' \rangle)}{(e \ \texttt{instanceof} \ C, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (\bot, \langle \sigma', \kappa' \rangle)} \quad \begin{array}{l} r_{C'} \in \textbf{Ref} \\ \kappa'(r) = \varphi_{C''} \\ C \neq C'' \end{array}$$

Table 7: New, Cast and InstanceOf

The only terms left to consider are casts, object creation and instanceof expressions. Once again, we need an auxiliary definition:

Definition 2.8 (New and Fresh) *Let* **init** *be the function, which maps a type to the initial value of this type, i.e. int to* 0*, bool to* $\perp$ *and any class type to* $\diamond$*. Then new is defined to be*

$$
\begin{aligned}
&new : \mathbf{ClassId} \to \mathbf{Object} \\
&new(C_0) \ := \ \varphi_{C_0}
\end{aligned}
$$

where

$$
\varphi(f, C) \ := \begin{cases} init(t) & if \ \ (fields_\Gamma \ C)(f) = (acm, t) \ \wedge \ C_0 = C \ \wedge \\ & (\ acm = \triangle \ \vee \ acm = C_0\ ) \\ & \\ init(t) & if \ \ (fields_\Gamma \ C)(f) = (\triangle, t) \ \wedge \ C_0 \neq C \ \wedge \\ & C_0 \preceq C \\ & \\ undefined & otherwise \end{cases}
$$

Let us furthermore say, that an adress r is **fresh** *with respect to a state* $\langle \sigma, \kappa \rangle$*, if and only if* $\forall x \in \mathbf{Id}.\sigma(x) \neq r$ *and* $r \notin dom(\kappa)$*.*

Freshness of an adress is best be thought of as allocating new space on the heap, just like `malloc` does in `C`. In the definition of *new*, we can once again recognize the concept of shadowing fields. If a new object of type C is created, all its fields have to be initialized by some initial values determined by *init*. All its fields means, all the fields declared in C as well as all the non-private fields in all classes C' with $C \preceq C'$. Here, we can have multiple declarations of the same field name in different classes, which are distinguished by their classes: ($\mathbf{FieldId} \times \mathbf{ClassId}$). Only one of these fields can be seen at a time, though.

With the help of these definitions, the rules in table 7 are easy to understand. New creates a C object on the heap and returns a C reference to it. Casts merely change the reference to an object, but not the object itself. Of course, there are some validity conditions on casts. Instanceof checks, whether a reference actually refers to an object of a certain type: the difference between reference and object types again.

Evaluation of a Program. Now, we are left to say, how a whole program is evaluated. This is basically just the evaluation of the statement following the class declarations in an initial state that we might assume empty for our purposes. One could think of some class libraries, too. The formal definition looks like this:

Definition 2.9 (Program Evaluation) *Let* (CT, s) *be a program and* Γ *the class environment associated with* CT. $\langle\sigma, \kappa\rangle$ *shall be some initial state. Then* $((CT, s), \langle\sigma, \kappa\rangle) \ \rightarrow_\Gamma \ (s, \langle\sigma, \kappa\rangle)$.

Sample Run of the Semantics. To conclude this section, we want to perform a run of the semantics on the example program given on page 14. We shall write $\rightarrow^\star_\Gamma$ to represent the reflexive, transitive closure of $\rightarrow_\Gamma$. We will ignore the outermost bind construct, which encloses the top level declarations in the main method, too. Γ shall be like shown on page 16 and references are enumerated like $r^1, r^2, \ldots$. Finite functions are represented as list of pairs. Because of the size of the run, we will only give an excerpt illustrating some nice features.

$$(\texttt{A a = new A();}\ \ D\, s, \langle[], []\rangle) \ \rightarrow^\star_\Gamma \ (D\, s, \langle[a \mapsto r^1_A], [r^1 \mapsto (A, [(a, A) \mapsto 0])]\rangle)$$

applying rules for local declarations and object creation.

$$\begin{array}{l}(\texttt{A ab = new B();}\ \ D\ s, \langle[a \mapsto r^1_A], [r^1 \mapsto (A, [(a, A) \mapsto 0])]\rangle) \\ \quad \rightarrow^\star_\Gamma \ (D\ s, \langle\sigma_1, \kappa_1\rangle)\end{array}$$

$$\begin{array}{l}\sigma_1 = [a \mapsto r^1_A,\ ab \mapsto r^2_A] \\ \kappa_1 = [r^1 \mapsto (A, [(a, A) \mapsto 0]),\ r^2 \mapsto (B, [(a, B) \mapsto 0, (a, A) \mapsto 0])]\end{array}$$

Note, that ab is an A reference to a B object, which has two A fields. Right now, only (a, A) can be seen, though. The next declaration does not reveal anything new, so we skip it. It should be clear how the state changes, i.e. b is bound to r^3_B and yet another B object appears on the heap. Let $\langle\sigma_2, \kappa_2\rangle$ be this new state. The first statement to be evaluated is method invocation on a.

$$\begin{array}{l}(\texttt{a.m();}\ \ s, \langle\sigma_2, \kappa_2\rangle) \\ \quad \rightarrow^\star_\Gamma \ (\texttt{bind}\ \langle[\texttt{this} \mapsto r^1_A], \kappa_2\rangle\ \texttt{in this.a = 1; return}\ \star;, \langle\sigma_2, \kappa_2\rangle)\end{array}$$

The rules for method invocation are applied here. According to the state, a evaluates to r^1_A. So the call to our lookup function *call* looks like

$$call(r^1, \langle\sigma_2, \kappa_2\rangle, m, \varepsilon)$$

yielding the bind-construct shown above. Now the assignment is evaluated in the state bound in the bind-construct. `This` is looked up, which yields r^1_A, as it is written in the local state. In the object located at r^1, the a field associated with class A (it is actually the only a field here) is then assigned 1, resulting in a heap, where at r^1 the object

$$(A, [(a, A) \mapsto 1])$$

is located. We return with value $\star$, which is ignored anyway. So the next method call to b will be performed in a state with components σ_2 and the new heap just described, because leaving the bind-construct results in dropping the first component of the local state and keeping the updates on the heap. The call `b.m();` is very similar and will update the B object at location r^3. The next method call is most interesting. It is performed in a state with first component σ_2 and the heap having two objects updated as described. Let us call this state $\langle\sigma_2, \kappa_3\rangle$. When we have looked up ab in σ_2 yielding r^2_A, we have another method lookup performed by *call*

$$\mathit{call}(r^2, \langle\sigma_2, \kappa_3\rangle, m, \varepsilon)$$

As $\kappa_3(r^2) = (B, \ldots)$, we start looking for methods in class B, hence we end up in the following configuration

$$\texttt{bind}\ \langle[\texttt{this} \mapsto r^2_B], \kappa_3\rangle\ \texttt{in this.a = 2; return}\ \star;$$

Note that r^2 has become a B reference now, thus we update the (a, B) field by evaluating the assignment. After we have returned the object located at r^2 looks like

$$(B, [(a, A) \mapsto 0,\ (a, B) \mapsto 2])$$

whereas ab is still mapped to r^2_A. This should be enough to understand what happens in the final statements. As described earlier, the first comparison fails, as ab still is an A reference. If it is casted down, it becomes a B reference and the a field of this is actually 2. This is dealt with in the semantic rule cast2.

$$(ab, \ldots)\ \rightarrow^\star_\Gamma\ (r^2_A, \ldots)$$

where r^2 actually points to a B object. Cast2 can be applied

$$(\texttt{(B)ab}, \ldots)\ \rightarrow_\Gamma\ (r^2_B, \ldots)$$

and the comparison `((B)ab).a == 2` becomes true.

3 A Control Flow Analysis for TJ

In this section, we will design a control flow analysis for TJ in the manner of [12], which was described in 1.2. The arguments from [12, chapter 3] made for a small functional calculus are adapted to TJ and equally proven. In the first part, we will specify abstract domains according to the semantic domains as presented in definition 2.1. Then we are going to specify the analysis by saying when a given analysis result is correct. This specification will then yield a constraint system, of which we shall show, that a solution always exists and can be computed. A semantic-based correctness result concludes this section.

3.1 A Syntax-Directed Specification

Extension with Labels. In the style of [12], we extend the TJ abstract syntax of expressions with *labels*. This means that expressions will be decorated with, say, natural numbers in order to state something about abstract values occurring at these labels. For the extended grammar, imagine every right-hand side occurence of *Expr* in the TJ sytnax replaced with *Exp* and the clause

$$Exp \ ::= \ Expr^l$$

added to the original grammar with l ranging over labels. Intuitively, there is a natural number attached to every expression. Note that the extension of the semantics from section 2 with labels is straightforward. Labels do not influence the progress of an evaluation but are merely carried around. In the course of this work, labels which are unneccesary and not used in certain contexts are omitted for the sake of readability. Now, we are able to define the abstract domains, which will be denoted $\hat{X}$, if X was the according concrete domain.

Definition 3.1 (Abstract Domains) *Let P be a program and* **Id**, **ClassId**, **FieldId** *and* **MethId** *like in definition 2.1. We define the following abstract domains as well as metavariables reaching over them.*

$$\begin{array}{rcll}
l & \in & \mathbf{Label} & \textit{a set of labels occuring in } P \\
x & \in & \widehat{\mathbf{Id}} & := \ \mathbf{Id} \times \mathbf{ClassId} \times \mathbf{MethId} \\
(C, C') & \in & \widehat{\mathbf{Ref}} & := \ \mathbf{ClassId} \times \mathbf{ClassId} \\
\hat{v} & \in & \widehat{\mathbf{Value}} & := \mathcal{P}(\widehat{\mathbf{Ref}}) \\
\hat{o}, \hat{\varphi} & \in & \widehat{\mathbf{Object}} & := \ \mathbf{FieldId} \to \widehat{\mathbf{Value}} \\
\langle \hat{\sigma}, \hat{\kappa} \rangle & \in & \widehat{\mathbf{State}} & := (\widehat{\mathbf{Id}} \to \widehat{\mathbf{Value}}) \ \times \ (\widehat{\mathbf{Ref}} \to \widehat{\mathbf{Object}}) \\
\hat{\zeta} & \in & \widehat{\mathbf{Cache}} & := \mathbf{Label} \to \widehat{\mathbf{Value}}
\end{array}$$

Labels shall be thought of as something like natural numbers. They are used to mark certain program points of interest. The abstract identifiers are nothing particular. We merely assume, that two occurrences of x in different methods should be distinguishable. This helps us making the analysis more precise and can be done statically while parsing. For simplicity, we shall use the same notation as in the semantics, while keeping in mind, that we have changed some identifier's names, in order to make them unique. Apart from this, abstract identifiers should be thought of as ordinary ones. The first major abstraction concerns references. We drop the notion of adresses completely. References with the same type referring to objects of the same type cannot be distinguished anymore. (C, C') stands for all references of type C pointing to an object of type C'. When we move from concrete to abstract values, we forget about primitive types and do only bother with abstract references. As the analysis has to be approximative, we need sets of abstract references to describe abstract values. This has to be understood in the following sense: Every concrete value (i.e. its abstract description), that can occur during run-time at a certain program point has to be included in the abstract value at this program point. The object type, which was included in an object's domain in the semantics, has been moved to abstract references. The abstract state accords to the concrete one using abstract domains instead of concrete ones, except for the way `this` is handled. It is now part of the cache and not of the identifiers anymore due to precision (`this` occurs in too many different contextes, which can be distinguished by their labels, though). $\widehat{\textbf{Cache}}$ deals with abstract values, that can occur at a certain program point which is marked by a label.

Eventually, we make some assumptions concerning programs: We consider Γ to be fixed for a given program, and it will always be used, even though it is not explicitely given. Also, we only allow one field dereferencing in a Var expression (several dereferencings can be resolved as syntactic sugar). In the specification part, we will drop the intermediate terms introduced in section 2.2. We will return to them later in the correctness part of this section.

The Acceptability Relation. We are now in a position to define the so-called *acceptability relation*, $\models$, between an analysis result, i.e. an abstract cache and an abstract state, and a program. This relation says whether an abstract cache and an abstract state correctly describe the control flow of a given program. In the definition, we need some auxiliary relations between analysis results and class declarations, method declarations, statements, local declarations and expressions respectively. Some of these relations have to be decorated with addtional information, in detail we have:

- $\models \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{Program}$
- $\models_c \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{ClassDecls}$
- $\models_m^C \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{Meths}$ decorated with a class name C used to perform method lookup.
- $\models_d^l \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{LocalDecls}$ decorated with a label representing the current `this` reference.
- $\models_s^l \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{Stms}$ decorated with a label, see above.
- $\models_e^l \subseteq \widehat{\textbf{Cache}} \times \widehat{\textbf{State}} \times \mathit{Exp}$ decorated with a label, see above.

In what follows we will apply infix notation for all of the above relations.

Definition 3.2 (Acceptability Relation) *The acceptability relation $\models$ is the largest relation satisfying the clauses in tables 8-11. If $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models P$ for a given program P, we shall call $\widehat{\textbf{Cache}} \times \widehat{\textbf{State}}$ a solution.*

$$
\begin{array}{rl}
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models & (CT, s) \\
\iff & (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_c CT \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^\infty s \\
\\
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_c & \texttt{class}\ C\ \texttt{extends}\ C'\ \{F\ M\} \quad CT \\
\iff & (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_m^C M \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_c CT \\
\\
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_c & \varepsilon \\
\iff & \text{always} \\
\\
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_m^C & acm\ t\ m(t_1\ x_1, \ldots, t_n\ x_n)\ \{s\} \quad M \\
\iff & \begin{array}{l} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_m^C M \quad \wedge \\ \forall l \in \gamma(m, C).(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s \end{array} \\
\\
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_m^C & \varepsilon \\
\iff & \text{always}
\end{array}
$$

Table 8: The Acceptability Relation for Programs

In table 8, the clauses for programs, class and method declarations are given. It is important to stress, that every method body in the class hierarchy is analyzed *exactly once*, thus syntax-directed analysis. This is done when analyzing the class hierarchy by predicting from which labels a method may be called. We therefore define a prediction function.

Definition 3.3 (Callabel Function) *Let (CT, s) be a program, $C \in$* **ClassId** *and $m \in$* **MethId**. *Then the callabel function γ is defined*

$$\gamma \;:\; \textbf{ClassId} \times \textbf{MethId} \rightarrow \; \mathcal{P}(\textbf{Label})$$

$$\gamma(C_0, m_0) \;:=\; \left\{ \begin{array}{ll} \{l \mid e^l.m_0(\ldots) \textit{ is in } C_0\} & \textit{if } C_0 = ((meths_\Gamma\ C_0)m_0)_{\downarrow 1} \\ \{l \mid e^l.m_0(\ldots) \textit{ is in } CT\} & \textit{otherwise} \end{array} \right.$$

Note that we are at a point where we lose precision. Certainly, γ will overestimate the places from which a function will actually be called at run-time. It simply returns all places, where the method name occurs, but respects accessability right. The approximation will be bad if we have lots of overwritten methods. In order to improve the analysis, one might start at this point by defining a subtler version of γ making use of more static information like type information. The set of labels resulting from a call to γ is the set of possible `this` references for a method. So, every method will be analyzed in the context of all possible `this` references, that might invoke the method, probably even more. However, this is neccessary for a syntax-directed specification and thus an inductive definition of an acceptability relation. Note that even unreachable code will be analzed in this approach. Eventually, the analysis of the statements belonging to the main method of a program is performed in the context of a ∞ label. This means a label that does not occur in the program and $\infty \notin$ **Label**, because there is no `this` reference in the main methods, which is not invoked by an object.

Table 9 is short but very interesting, because it demonstrates the interplay between reference and object types in terms of our analysis. For every pair of reference and object type, that might be included in the assigned expression, the variable to be assigned can have the abstract value consisting of the *declared* type paired with the object type derived from analyzing the expression.

Most of the clauses in table 10 are simple recursive traversals of the program. The only places of interest are assignments. If we assign an expression e^{l_0} to a variable x, all abstract values, that e might have, have to be included in the possible values for x. If we have `this` on the left-hand side of an

$$
\begin{array}{lll}
(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \models^{l}_{d} & t\ x\ =\ e^{l_0} \quad decls & \\
& (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \models^{l}_{d}\ decls \quad \wedge & \\
\iff & (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \models^{l}_{e}\ e^{l_0} \quad \wedge & \\
& \forall (C, C') \in \hat{\zeta}(l_0).\{(t, C')\} \subseteq \hat{\sigma}(x) & \\
& \text{if } t \in \mathbf{ClassId} & \\
& & \\
(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \models^{l}_{d} & \varepsilon & \\
\iff & \text{always} &
\end{array}
$$

Table 9: The Acceptability Relation for Local Declarations

assignment, the abstract values have to be included in the possible values of `this`. Which `this`? The one labelling the relation. It is a label that was set at some method invocation. Here, it is important that the representation of the `this` reference has been moved to the abstract cache. It is not part of the abstract state. Assigning to fields makes use of the abstract heap $\hat{\kappa}$. Each abstract value that x or `this` might have is dereferenced, and the values of e are included in the corresponding field for each of these abstract objects. Note that our notation only allows one field dereferencing because of simplicity. Otherwise, we would have had to define another relation for variables only.

As with the expression clauses defined in table 11, constants and operators are either trivial or recursive traversals. The constraints on variables and field dereferencing correspond exactly to the ones used for assignments, except for left-hand side and right-hand side being swapped. The `new` clause simply includes a (C, C) abstract value in the cache value of the label `new` is labelled with. Instanceof is another recursive traversal, even though our analysis could be used to predict the result of an instanceof expression. However, the result is of boolean type, thus we cannot use it furtherly. Another extension would be to include data flow analysis fragments, thus being able to use the information obtained from the instanceof clause. Casts resemble the clause for local variable declarations. The reference type component of all abstract values at label l_0 are changed to the cast type at label l_1. Method calls are handled like this: First, we have to have correct analysis results for all arguments and the calling expression. Then, we look up which abstract values the receiver object can have, thus determining which functions there can be called from here. After that, we require, that all possible values of the

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{skip}$$
$$\iff \text{always}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{if}\ e^{l_0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$
$$\iff \begin{array}{l} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s_1 \quad \wedge \\ (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s_2 \end{array}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s_1;\ s_2$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s_1 \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s_2$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \{\ decl\ s\ \}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_d^l decl \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l x := e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad \hat{\zeta}(l_0) \subseteq \hat{\sigma}(x)$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{this} := e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad \hat{\zeta}(l_0) \subseteq \hat{\zeta}(l)$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l x.f := e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad \forall \hat{r} \in \hat{\sigma}(x). \hat{\zeta}(l_0) \subseteq ((\hat{\kappa}\ \hat{r}) f)$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{this}.f := e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad \forall \hat{r} \in \hat{\zeta}(l). \hat{\zeta}(l_0) \subseteq ((\hat{\kappa}\ \hat{r}) f)$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{return}\ e^{l_0}$$
$$\iff (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{return}\ \star$$
$$\iff \text{always}$$

Table 10: The Acceptability Relation for Statements

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad \mathit{const}$$
$$\iff \quad \text{always, if } \mathit{const} \in \mathbf{B} \cup \mathbf{Z} \cup \{\texttt{null}\}$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad e_1 \ \mathit{bop} \ e_2$$
$$\iff \quad (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e_1 \quad \wedge \quad (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e_2$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad \mathit{uop} \ e$$
$$\iff \quad (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad x^{l_0}$$
$$\iff \quad \hat{\sigma}(x) \subseteq \hat{\zeta}(l_0)$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad \texttt{this}^{l_0}$$
$$\iff \quad \hat{\zeta}(l) \subseteq \hat{\zeta}(l_0)$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad x.f^{l_0}$$
$$\iff \quad \forall \hat{r} \in \hat{\sigma}(x).((\hat{\kappa} \ \hat{r})f) \subseteq \hat{\zeta}(l_0)$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad \texttt{this}.f^{l_0}$$
$$\iff \quad \forall \hat{r} \in \hat{\zeta}(l).((\hat{\kappa} \ \hat{r})f) \subseteq \hat{\zeta}(l_0)$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad \texttt{new} \ C^{l_0}$$
$$\iff \quad \{(C,C)\} \subseteq \hat{\zeta}(l_0)$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad e \ \texttt{instanceof} \ C$$
$$\iff \quad (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad (C)e^{l_0 \ l_1}$$
$$\iff \quad \forall (C',C'') \in \hat{\zeta}(l_0).\{(C,C'')\} \subseteq \hat{\zeta}(l_1) \quad \wedge \quad (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e^{l_0}$$

$$(\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \quad e_0^{l_0}.m(e_1^{l_1},\ldots,e_n^{l_n})^{\ l'}$$
$$\iff \quad \begin{array}{l} (\hat{\zeta},\langle\hat{\sigma},\hat{\kappa}\rangle) \models^l_e \ e_i^{l_i} \quad i = 0 \ldots n \quad \wedge \\ \forall (C,C') \in \hat{\zeta}(l_0) \forall i = 1 \ldots n. \ \hat{\zeta}(l_i) \subseteq \hat{\sigma}(x_i) \ \wedge \ \hat{\zeta}(l'') \subseteq \hat{\zeta}(l') \\ \text{where } \mathit{mlookup}(C,m) = (\mathit{acm}, t, t_1 \ x_1, \ldots, t_n \ x_n, s, e'') \\ \qquad \wedge \quad t \neq \texttt{void} \end{array}$$

Table 11: The Acceptability Relation for Expressions

arguments are included in the formal parameters of the methods we found and that the return values of the methods are included in the cache value of the label that marks the whole method call expression. Note that method bodies are not analyzed at this point. This was done when analyzing the class hierarchy. Also, the return expressions have been analyzed there, so we only need to include these results in the results of the method call.

3.2 Theoretical Properties

The first important observation is, that the relation defined in 3.2 indeed is a relation, i.e. well-defined. This is clear from the inductive, thus syntax-directed, definition. The fact that method bodies are analyzed exactly once is essential for this proposition. Apart from this, there are more properties, which can be formally proven. We will begin by checking, whether for a given program, there always exists a solution, i.e. a relation satisfying all of the above clauses. We need to apply some lattice theory in order to prove this. Secondly and equally important, the link between the semantics and the analysis is made by means of a subject-reduction result. It says that a correct analysis remains correct under evaluation. Eventually, we will hint to how analysis results can be obtained with the help of a constraint-based formulation.

3.2.1 Existence of Solutions

We assume some familiarity with the concept of complete lattices along with some basic properties. Further information can be found in various textbooks on this topic such as [5]. It will be shown, that the domain of our analysis actually is a complete lattice and that the set of solutions is a *Moore family* of this lattice. The properties of Moore families will then be exploited to obtain an existence result.

Lemma 3.1 *Let P be a program. Then the set* $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ *obtained from P is a finite complete lattice.*

Proof. The proof is constructive, i.e. the ordering will be developped during the proof. Note, that all our domains are finite, in particular **ClassId**, **FieldId**, $\widehat{\mathbf{Id}}$ and **Label** are finite, because they are made up of class names, field names, variable names and labels occuring in the P. Hence $\widehat{\mathbf{Ref}}$ is finite as the cartesian product of **ClassId** with itself. $\widehat{\mathbf{Value}}$ is finite being the powerset of the finite set $\widehat{\mathbf{Ref}}$. It can be canonically made a complette lattice applying subset ordering. $\widehat{\mathbf{Id}} \to \widehat{\mathbf{Value}}$, $\widehat{\mathbf{Ref}} \to \widehat{\mathbf{Value}}$ and $\widehat{\mathbf{Label}} \to \widehat{\mathbf{Value}}$ can thus be constructed as the *total function space* from $\widehat{\mathbf{Id}}$, $\widehat{\mathbf{Ref}}$ and $\widehat{\mathbf{Label}}$

respectively to $\widehat{\mathbf{Value}}$. The total function space from a set S to a complete lattice $(L, \subseteq)$ is defined to be the set of total functions from S to L with the following ordering $\sqsubseteq$: For $f_1, f_2 \in S \to L$

$$f_1 \sqsubseteq f_2 \iff \forall s \in S. f_1(s) \subseteq f_2(s)$$

Note that the functions in our analysis indeed are total, or can be made total by mapping unused variables, abstract references or labels to the empty set of abstract values. This will be in particular employed for variables denoting non-object types, which do not occur in our analysis. Finally, the three total function spaces can be combined to the cartesian product with component-wise ordering, i.e.

$$(\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1 \rangle) \sqsubseteq (\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2 \rangle) \iff \hat{\zeta}_1 \sqsubseteq \hat{\zeta}_2 \ \wedge \ \hat{\sigma}_1 \sqsubseteq \hat{\sigma}_2 \ \wedge \ \hat{\kappa}_1 \sqsubseteq \hat{\kappa}_2$$

The meet and join operations on $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ can be derived from ordinary set intersection and set union lifted to the complete domain in the obvious way. We can conclude, that $(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)$ indeed is a finite, complete lattice with the ordering constructed in the proof. $\Box$

Definition 3.4 (Moore Family) *A Moore family is a subset of a complete lattice $(L, \subseteq)$, that is closed under greatest lower bounds.*

It follows immediately from the definition, that a Moore family is never empty, because the greatest lower bound of the empty set is the whole complete lattice. The whole lattice is thus also included in a Moore family. Furthermore, a Moore family always has a least element. Formal proofs of these claims are trivial, hence omitted.

Lemma 3.2 *Let P be a program. Then the set*

$$S_P \ := \ \{ \ (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \in \widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}} \ \mid \ (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \ \models \ P \ \}$$

is a Moore family.

Proof. We shall write $(\hat{\zeta}^\top, \langle \hat{\sigma}^\top, \hat{\kappa}^\top \rangle)$ for the top element of $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$, i.e. the triple of functions, that map each label, identifier and abstract reference to the complete set of abstract values. Now we shall prove

$$(\hat{\zeta}^\top, \langle \hat{\sigma}^\top, \hat{\kappa}^\top \rangle) \in S_P \tag{1}$$

$$\begin{aligned} &(\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1 \rangle) \in S_P, \ (\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2 \rangle) \in S_P \\ &\qquad \Rightarrow (\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1 \rangle) \ \sqcap \ (\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2 \rangle) \ \in S_P \end{aligned} \tag{2}$$

where $\sqcap$ denotes the meet operation on $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$. (1) is clear, because we only have set inclusion constraints in the specification. $(\hat{\zeta}^\top, \langle \hat{\sigma}^\top, \hat{\kappa}^\top \rangle)$ means that on every right-hand side, there is the complete set of abstract values, which makes all these set constraints true. So, S_P is not empty. As with (2), we have to apply structural induction on the parse tree of a program. The single cases are very simple due to the form of our constraints. We shall merely apply the following fact about arbitrary sets A, B, C, D

$$\left.\begin{array}{c} A \subseteq B \\ C \subseteq D \end{array}\right\} \Rightarrow \; A \cap C \subseteq B \cap D$$

All the cases basically reduce to this fact, as becomes clear from closer inspection. To conclude the proof, we observe that every subset $Y$of S_P is finite, because $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ is finite according to lemma 3.1. Hence, the greatest lower bound of Y is the finite intersection of its elements. By induction on the size of Y, it follows from (2) that S_P indeed is a Moore family. $\square$

From the above lemmas as well as from the comments made about the properties of Moore families, we immediately get the following existence result, where least is to be understood in terms of the ordering of $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$.

Theorem 1 (Existence of Least Solutions) *Let P be a program. Then there exists a least solution $(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \in \widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ satisfying*

$$(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \;\models\; P$$

3.2.2 Semantic Correctness

We have shown, that least solutions always exist for a given program. Now, we want to link these solutions to the semantics from section 2. We want to obtain a subject reduction result, saying that a solution remains correct under evaluation. To do so, we have to make some preparations. As there are intermediate terms appearing during evaluation, we have to extend the acceptability relation to these intermediate terms. Furhtermore, we need some relations between concrete domains of the semantics and abstract domains of the analysis, that tell whether an abstract domain is a valid description or abstraction from the concrete one. Let us first define these correctness relations.

Definition 3.5 (Correctness Relations) *Be $\langle \sigma, \kappa \rangle \in$ **State**, $\langle \hat{\sigma}, \hat{\kappa} \rangle \in \widehat{\mathbf{State}}$, $v \in$ **Value**, $\hat{v} \in \widehat{\mathbf{Value}}$, $C \in$ **ClassId**, $\varphi_{C'} \in$ **Object** and*

$\hat{\varphi} \in \widehat{\mathbf{Object}}$. *The relations*

$$\begin{aligned}
\mathcal{S} &\subseteq \mathbf{State} \times \widehat{\mathbf{State}} \\
\mathcal{V} &\subseteq \mathbf{Value} \times \widehat{\mathbf{Value}} \\
\mathcal{O} &\subseteq (\mathbf{Adr} \rightarrow \mathbf{Value}) \times \widehat{\mathbf{Object}}
\end{aligned}$$

are defined mutually recursive as follows

$$\begin{aligned}
&\Big(\langle \sigma, \kappa \rangle, \langle \hat{\sigma}, \hat{\kappa} \rangle\Big) \;\in\; \mathcal{S} \\
&\iff \begin{cases} \forall x \in \mathbf{Id}.\Big(\sigma(x), \hat{\sigma}(x)\Big) \in \mathcal{V} \quad \wedge \\ \forall r_C \in \mathbf{Ref}.r_c \textit{ not fresh},\ \kappa(r) = \varphi_{C'},\ \hat{\kappa}(C, C') = \hat{\varphi} \\ \qquad \Rightarrow\ \Big(\varphi, \hat{\varphi}\Big) \in \mathcal{O} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\Big(v, \hat{v}\Big) \;\in\; \mathcal{V} \\
&\iff \begin{cases} v \textit{ is undefined} \quad \vee \\ v \in \mathbf{B} \cup \mathbf{Z} \cup \{\diamond\} \cup \{\star\} \\ v = r_c,\ \kappa(r_c) = \varphi_{C'} \ \Rightarrow\ (C, C') \in \hat{v} \end{cases} \\
&\textit{for a fixed state } \langle \sigma, \kappa \rangle
\end{aligned}$$

$$\begin{aligned}
&\Big(\varphi, \hat{\varphi}\Big) \;\in\; \mathcal{O} \\
&\iff \forall (f, C) \in \mathit{dom}(\varphi).\Big(\varphi(f, C), (\hat{\varphi}\ f)\Big) \in \mathcal{V} \\
&\textit{for a fixed state } \langle \sigma, \kappa \rangle
\end{aligned}$$

For the rest of this work we shall use infix notation as before. Strictly speaking, the fixed state used in the definition of $\mathcal{V}$ and $\mathcal{O}$ should be included in the relation. As this state will always be clear from the context, we omit it for the sake of readability. The condition that r_C must not be fresh is a very technical one. Neither r_C not in the image of σ nor r_C in the domain of κ are sufficient. The domain of κ consists of adresses, whereas we need references; and the image of σ is not sufficient for the `new` case of the proof. Freshness comprises both concepts. The relations should be interpreted like this: an abstract state, value or object is a valid description of a concrete one. Note that we are only interested in object type values in the case of $\mathcal{V}$. Furthermore, it is obvious that the relations are well-defined. Whether an abstract and a concrete value belong to $\mathcal{V}$ can immediately be decided. $\mathcal{O}$ makes use of $\mathcal{V}$ only, namely for a finite number of field names. Finally, $\mathcal{S}$ only checks a finite number of values and objects.

Analyzing Intermediate Terms. As intermediate terms appear under evaluation, and as we are interested in a subject reduction result, we have to extend the acceptability relation to intermediate terms. The extended cases are given in table 12.

The analysis of intermediate terms needs the correctness relation defined

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l \texttt{bind } \langle\sigma, \kappa\rangle^{l'} \texttt{ in } \{decls\ s\}$$
$$\iff \begin{array}{l} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^{l'} s \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_d^{l'} decls \\ \langle\sigma, \kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma}, \hat{\kappa}\rangle \quad \wedge \quad \sigma(\texttt{this})\ \mathcal{V}\ \hat{\zeta}(l') \end{array}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l r_C.f\ := e^{l_0}$$
$$\iff \begin{array}{l} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e^{l_0} \quad \wedge \quad \hat{\zeta}(l_0) \subseteq (\hat{\kappa}(C, C')f) \\ \text{where } \kappa(r) = \varphi_{C'} \end{array}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l v^{l_0}$$
$$\iff v^{l_0}\ \mathcal{V}\ \hat{\zeta}(l_0)$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l r_C.f^{l_0}$$
$$\iff \begin{array}{l} (\hat{\kappa}(C, C')f) \subseteq \hat{\zeta}(l_0) \\ \text{where } \kappa(r) = \varphi_{C'} \end{array}$$

Table 12: The Acceptability Relation for Intermediate Terms

above. Labelled expressions keep their labels, even after they have been evaluated to a value. Also, we make implicit use of a semantic state, which was dropped for notation reasons. Again, it will be clear from the context, which state is actually meant, because analysis of intermediate terms only occurs during evaluation, thus in a semantic context. A forthcoming proof will clarify this situation.

Note that states within a bind-construct are annotated with labels, too. This is neccessary to keep track of local `this` references. The recursive descent in a bind-construct is e.g. performed in the context of l' instead of l. There are two possibilities of entering a bind-construct: evaluating a local declaration or a method call. In the first case of a local declaration, it is clear that the `this`-reference does not change on entering the bind construct. To formally define the label-extended version of the semantical rule [bind], we would also need to annotate $\rightarrow_\Gamma$ with a label denoting the current `this`-reference. That would make notation more unreadable and would be used only once. Intuitevely, it should be clear how this extension could be defined formally. Consider also the [bind] part of the forthcoming proof for clarification.

If a bind-construct is entered via a method call, the additional labels are added by an extended version of the *call* function, which was defined in 2.7. The extended version is denoted $\widehat{call}$ and shall be applied instead of the original version. The definition is exactly like in 2.7 except for the defining clause, where labels are added at some points.

Definition 3.6 *Given the conditions of definition 2.7 with label-extended syntax. Define*

$$\begin{array}{l} \widehat{call}(r_C^{l_0}, \langle\sigma,\kappa\rangle, m, v_1^{l_1}, \ldots, v_n^{l_n})^l \;:=\; \boldsymbol{bind}\ \langle\sigma',\kappa\rangle^{l_0}\ \boldsymbol{in}\ \{s\ \boldsymbol{return}^l\ e^{l'};\} \\ \text{where } \sigma' = [\boldsymbol{this} \mapsto r_C^{l_0}][x_i \mapsto v_i^{l_i}]_{i=1}^n \end{array}$$

Make yourself clear, that all extensions added in order to perform the analysis or for technical and proof reasons have no influence on the runs of the original semantics. There is only a little more notation introduced, that would be merely carried around in the semantics. This would distract from the important points of the semantics and has thus been dropped.

The Correctness Result. All the tools and vocabulary needed to express a correctness result are now at hand. The result will first be given and its meaning explained. In the rest of this section, the proof will be carried out.

Theorem 2 (Semantic Correctness) *Be (CT, s) a program and Γ the induced class environment. Let $(\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \in \widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ be such that $(\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \models (CT, s)$. Then the following holds for all configurations $(t, \langle\sigma,\kappa\rangle), (t', \langle\sigma',\kappa'\rangle) \in \mathbf{Config}$ where $t \neq (CT, s)$. If*

$$\begin{array}{ll} (t, \langle\sigma,\kappa\rangle) \;\to_\Gamma\; (t', \langle\sigma',\kappa'\rangle) \;\wedge & (C1) \\ (\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \;\models_t^l\; t \;\wedge & (C2) \\ \langle\sigma,\kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle \;\wedge & (C3) \\ \sigma(\boldsymbol{this})\ \mathcal{V}\ \hat{\zeta}(l) & (C4) \end{array}$$

then holds

$$\begin{array}{ll} (\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \;\models_{t'}^l\; t' \;\wedge & (R1) \\ \langle\sigma',\kappa'\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle \;\wedge & (R2) \\ \sigma'(\boldsymbol{this})\ \mathcal{V}\ \hat{\zeta}(l) & (R3) \end{array}$$

In the case that an evaluation terminates, i.e. yields a state only, the theorem also holds with (R2) and (R3) being the only conclusions to be drawn.

Some further explanation: The annotation of $\models$ with terms shall denote the clause that matches this term, like e for expressions, s for statements or d for declarations. Close inspection of the semantic rules reveals, that the three cases just mentioned actually are the only ones, which possibly can occur. This is due to the fact that these are the only constructs, where evaluation takes place. Other terms are dealt with by means of some auxiliary functions. Only the very first evaluation step, which maps a program to the statement in the main method, is an exception to this observation, hence excluded. It is also justified to annotate $\models$ with labels in all cases, because $\models_e$, $\models_s$ and $\models_d$ are also annotated with labels. In the formulation of the theorem, it becomes clear, what was meant by "the state will be clear from the context", as with respect to the correctness relations: $\mathcal{V}$ in (C4) refers to $\langle\sigma, \kappa\rangle$ and $\mathcal{V}$ in (R3) refers to $\langle\sigma', \kappa'\rangle$, which is both times the only state at hand.

What is the intuition behind this theorem, and does it actually establish correctness in some sense? Obviously, it expresses a *subject reduction* result, meaning that correct analyses remain correct during evaluation. In the case of a terminating program, this implies inductively, that our analysis result correctly describes all the actual configurations at all times – with respect to our correctness relations. So if we believe in the abstractions made, and if we believe that the abstractions are carefully chosen and a good way of modelling the configurations in a feasible way, then theorem 2 is indeed a correctness result. The fact that we exclude the start configuration is neglectable and technical. From definition 2.9 and the acceptability relation clause for programs it is clear, that the very first execution step satifies theorem 2, iff the initial state is in $\mathcal{S}$ relation with $\langle\hat{\sigma}, \hat{\kappa}\rangle$. We can easily impose this assumption, as the intital state will typically be empty or consisting of some bindings introduced by a library, which we may assume valid in terms of $\mathcal{S}$. The case of the start configuration has only been dropped in the theorem to make its formulation easier.

The Proof. The rest of this section deals with proving theorem 2. The proof applies structural induction on the shape of the inference tree used to establish $(t, \langle\sigma, \kappa\rangle) \ \rightarrow_\Gamma \ (t', \langle\sigma', \kappa'\rangle)$. Hereby, we will prove the cases of a configuration only consisting of a state and a configuration having both state and term simultaneously. Also, we will not distinguish between statements, expressions and so on. Let us start with some obvious facts.

Fact 3.1 *Given a program (CT, s), the induced class environment Γ and a solution $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \ \models \ (CT, s)$. Then*

$$\forall C \in \textbf{ClassId} \forall m \in \textbf{MethId} \forall l \in \gamma(C, m). \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \ \models^l_s \ s \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \ \models^l_e \ e^{l_0} \quad \textit{(CLP)}$$

where s and e^{l_0} result from $mlookup(C, m)$. This property shall be called the call-label-property (CLP).

This is obvious from the definition of $\models$ in case of class and method declaration. It could be formally shown by structural induction on programs. Fact 3.1 expresses that every method body – even if unreachable – is analyzed exactly once. (CLP) follows immediately from the assumptions made in theorem 2. Throughout the whole proof we shall assume (CLP).

Fact 3.2 *Be $v \in$ **Value** and $\hat{v}_1, \hat{v}_2 \in \widehat{\mathbf{Value}}$. If $v\ \mathcal{V}\ \hat{v}_1$ and $\hat{v}_1 \subseteq \hat{v}_2$ then $v\ \mathcal{V}\ \hat{v}_2$, where both $\mathcal{V}$ are interpreted with respect to the same state.*

An immediate corollary from the definition of $\mathcal{V}$ in 3.5. If an abstract reference is included in a set, then it is also included in every superset.

Fact 3.3 *Let $\langle\sigma,\kappa\rangle, \langle\sigma',\kappa'\rangle \in$ **State** and $\langle\hat{\sigma},\hat{\kappa}\rangle \in \widehat{\mathbf{State}}$. If $dom(\kappa) \subseteq dom(\kappa')$ (1) and $\forall r \in \mathbf{Adr}.\kappa'(r) = \varphi_C \Rightarrow \varphi\ \ O\hat{\varphi}$ (2) and $\langle\sigma,\kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle$ (3) then $\langle\sigma,\kappa'\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle$.*

This fact could be called a lemma, too, as it requires at least some explanation. From (2) of the fact we get immediately the second condition in the definition of $\mathcal{S}$ in the case of $\langle\sigma,\kappa'\rangle$. We are left to show for any variable $x \in \mathbf{Id}$ that $\sigma(x)\ \mathcal{V}\ \hat{\sigma}(x)$ where $\mathcal{V}$ is with respect to $\langle\sigma,\kappa'\rangle$. Without loss of generality let $\sigma(x) = r_C$ (if it is a basic value, it is trivial). Then we know from (3) that $\kappa(r) = \varphi_{C'}$ and $(C, C') \in \hat{\sigma}(x)$. The crucial observation is, that because of (1), $\kappa'(r)$ is defined and it yields $\varphi'_{C'}$, which suffices to establish the claim. This is because the type of an object residing at a given adress is created at a new-expression and never changed later on. Check all the semantic rules to see that this holds.

Fact 3.4 *Given a program P and a solution $(\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \models P$. Then for every expression $e^{l_0}.m(\ldots)^{l_1}$ and for every* **`return`** *e^{l_2}, in which such a method call can result at run-time, we have $\hat{\zeta}(l_2) \subseteq \hat{\zeta}(l_1)$.*

This is clear from the specification of the acceptability relation in the case for method calls, where return value passing happens.

Fact 3.5 *If $v\ \ V^{\langle\sigma,\kappa\rangle}\ \hat{v}$, i.e. $\mathcal{V}$ with respect to a state $\langle\sigma,\kappa\rangle$ and if $\langle\sigma',\kappa'\rangle$ is such that $dom(\kappa) \subseteq dom(\kappa')$, then $v\ \mathcal{V}^{\langle\sigma',\kappa'\rangle}\ \hat{v}$.*

Note that the only interesting case involves $v \equiv r_C$ due to the definition of $\mathcal{V}$. Now we can apply the same argumentation about κ components of states as in the proof of fact 3.3. The σ component is irrelevant here, because we have already evaluated an expression until only a value is left.

The Base Cases. Here are the base cases of the proof in the same order as they were introduced in section 2.

[*id*] The assumptions yield

$$
\begin{aligned}
(C1) &\Rightarrow (x_0^{l_0}, \langle\sigma,\kappa\rangle) \rightarrow_\Gamma (\sigma(x_0)^{l_0}, \langle\sigma',\kappa'\rangle) \\
(C2) &\Rightarrow \begin{cases} \hat{\zeta}(l) \subseteq \hat{\zeta}(l_0) & \text{if } x \equiv \texttt{this} \\ \hat{\sigma}(x_0) \subseteq \hat{\zeta}(l_0) & \text{otherwise} \end{cases} && (3) \\
(C3) &\Rightarrow \sigma(x_0^{l_0})\ \mathcal{V}\ \hat{\sigma}(x) && (4) \\
(C4) &\Rightarrow \sigma(\texttt{this})\ \mathcal{V}\ \hat{\zeta}(l) && (5)
\end{aligned}
$$

(R2) follows from (C3) as the state does not change. Showing (R1) means showing $\sigma(x_0^{l_0})\ \mathcal{V}\ \hat{\zeta}(l_0)$. For x_0 being a variable this follows from (4), (3) and fact 3.2. If $x_0 \equiv \texttt{this}$ we need to apply fact 3.2 to (5) and (3). (R3) is again equivalent to (C4) as l and $\langle\sigma,\kappa\rangle$ do not change.

[*access*2] (R2) and (R3) follow immediately from (C3) and (C4) as state and `this` reference are not changed by this rule. (C1) and (C2) yield

$$
\begin{aligned}
(C1) &\Rightarrow (r_{C_0}.f_0^{l_0}, \langle\sigma,\kappa\rangle) \rightarrow_\Gamma (\varphi(f_0, C_0)^{l_0}, \langle\sigma,\kappa\rangle) \\
&\quad\ \text{where } \kappa(r) = \varphi_{C'} \\
(C2) &\Rightarrow (\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle) \models_e^l r_{C_0}^{l_0} \\
&\Rightarrow \hat{\kappa}(C_0, C')f \subseteq \hat{\zeta}(l_0) && (6) \\
&\quad\ \text{where } C' \text{ is the object type of } \kappa(r) \\
& && (7)
\end{aligned}
$$

Showing (R1) means showing $\varphi(f_0, C_0)\ \mathcal{V}\ \hat{\zeta}(l_0)$. Setting $C = C_0$ and $f = f_0$ we get from the definition of the correctness relations for states

$$
\begin{aligned}
(C3) &\Rightarrow \varphi_{C'}\ \mathcal{O}\ \hat{\kappa}(C_0, C') \\
(\text{definition of } \mathcal{O}) &\Rightarrow \varphi(f_0, C_0)\ \mathcal{V}\ \hat{\kappa}(C_0, C')f_0 && (8)
\end{aligned}
$$

which together with 6 and fact 3.2 yields (R1). Note that r_{C_0} cannot be fresh here, so we may apply the definition of $\mathcal{R}$ for r_{C_0}

[*skip*] trivial

[*assign*3] Because of (C1) we have $(x_0 := v^{l_0}, \langle\sigma,\kappa\rangle) \rightarrow_\Gamma \langle\sigma[x_0 \mapsto v^{l_0}], \kappa\rangle$. There is nothing to show for (R1) here, because we do not demand anything for a state being a correct analysis result . Intuitively, it should exactly be (R2), which is to be proven now. Due to (C3) and the fact, that κ remains unchanged and σ is only updated for x_0, it suffices to prove the first condition in the definition of $\mathcal{S}$ for x_0, namely $\sigma[x_0 \mapsto v^{l_0}]\ (x_0) \ = \ v^{l_0}\ \mathcal{V}\ \hat{\sigma}(x_0)$. As we have $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models^l_s x_0 := v^{l_0}$ from (C2), we can conclude $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models^l_e v^{l_0}$ from the specification of $\models$ and hence $v^{l_0}\ \mathcal{V}\ \zeta(l_0)$. On the other hand, we get $\hat{\zeta}(l_0) \subseteq \hat{\sigma}(x_0)$ from (C2) and can thus apply fact 3.2 yielding (R2). (R3) follows immediately from (C4) if $x_0 \not\equiv \texttt{this}$. Otherwise, we have to show $v^{l_0}\ \mathcal{V}\ \hat{\zeta}(l)$. This follows from (C2) for $x_0 \equiv \texttt{this}$, i.e. $\hat{\zeta}(l_0) \subseteq \hat{\zeta}(l)$, exactly like in the case for $x_0 \not\equiv \texttt{this}$.

[*assign4*] (C1) means $(r_C.f_0 := v^{l_0}, \langle\sigma, \kappa\rangle) \ \rightarrow_\Gamma \ \langle\sigma, \kappa[r \mapsto o]\rangle$ where $o = (\varphi[(f_0, C) \mapsto v^{l_0}])_{C'}$, if $\kappa(r) = \varphi_{C'}$. Like above, there is nothing to show for (R1). As with (R2), it is obvious, that – due to the independence of σ and κ concerning the correctness relations – it suffices to show the second condition of $\mathcal{S}$ for only one reference, namely r_C. All the rest follows from (C3). We are left to show $\varphi[(f_0, C) \mapsto v^{l_0}]\ \mathcal{O}\ \hat{\kappa}(C, C')$. The only difference between the old φ and the updated one is at place (f_0, C). From the definition of $\mathcal{O}$, we get that it suffices to show $\varphi[(f_0, C) \mapsto v^{l_0}]\ (f_0, C) = v^{l_0} \ \ \mathcal{V}\ \hat{\kappa}(C, C')f_0$. Remember that $\mathcal{V}$ depends on a state. It is clear though, that v^{l_0} is not affected by these changes, regardless whether we consider $\mathcal{V}$ with respect to the old or new state (because $\mathcal{V}$ does not depend on a κ). Like before, (R3) is trivial if $\sigma(\texttt{this} \neq r_C)$. If this is the case

$$
\begin{aligned}
(C2) \ &\Rightarrow \ (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models^l_s r_C.f_0 := v^{l_0} \\
&\Rightarrow \ v^{l_0}\ \mathcal{V}\ \hat{\zeta}(l_0) \quad \wedge \qquad (9) \\
&\quad\ \ \hat{\zeta}(l_0) \subseteq \hat{\kappa}(C, C')f_0 \qquad (10)
\end{aligned}
$$

(9) and (10) together with fact 3.2 yield (R2).

[*nodecl*] trivial

[*bind*] Let l denote the label of the $\texttt{this}$-reference on entering the bind construct. According to the comments we made above, (C1) then yields

$$(\{decls\ s\}, \langle\sigma, \kappa\rangle) \ \rightarrow_\Gamma \ (\texttt{bind}\ \langle\sigma, \kappa\rangle^l\ \texttt{in}\ \{decls\ s\}, \langle\sigma, \kappa\rangle)$$

Note that we annotated the local state with the current label of $\texttt{this}$. Obviously, (R2) and (R3) follow from (C3) and (C4) as nothing changes concerning state and $\texttt{this}$. In order to establish (R1) we need to show $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models^l_s s$, $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models^l_d decls$, $\langle\sigma, \kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma}, \hat{\kappa}\rangle$ and

$\sigma(\texttt{this})\ \mathcal{V}\ \hat{\zeta}(l)$ – according to the analysis specification of a bind-construct. The first two conditions follow immediately from (C2), the third from (C3) and the fourth from (C4). This is not as trivial as it seems to be on first sight. It is crucial to note, that these four conditions refer to the state bound in the bind-construct and to the local `this` label l. In this particular rule, these local state and label just happen to be exactly the same as the global ones. This is different in case of method calls.

$[constn, constb1, constb2]$ trivial, due to the $\mathcal{V}$ relation neglecting basic values

$[call]$ Let

$$t \ \equiv\ r_C^{l_0}.m(v_1^{l_1} \dots v_n^{l_n})^{l'}$$

in (C1),

$$\texttt{bind}\ \langle\sigma', \kappa\rangle^{l_0}\ \texttt{in}\ \{s\ \texttt{return}\ e^{l''}\}\ :=\ \widehat{call}(r_C^{l_0}, \langle\sigma, \kappa\rangle, m, v_1^{l_1}, \dots, v_n^{l_n})$$

where

$$\sigma'\ :=\ [\texttt{this} \mapsto r_C^{l_0}][x_i \mapsto v_i^{l_i}]_{i=1}^n$$

The x_i are the formal parameters of the method found by $\widehat{call}$. Consider definitions 2.7 and 3.6 on how to determine these. From (C1) we then get the following

$$(t, \langle\sigma, \kappa\rangle)\ \rightarrow_\Gamma\ (\texttt{bind}\ \langle\sigma', \kappa\rangle^{l_0}\ \texttt{in}\ \{s\ \texttt{return}\ e^{l''}\}, \langle\sigma, \kappa\rangle)$$

which immediately yields (R2) and (R3) from (C3) and (C4) as global state and `this` again do not change. We still need to check these conditions for the local state. This is hidden in the requirements for analyzing bind-constructs, that is in (R1). We are left to show

$$\langle\sigma', \kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma}, \hat{\kappa}\rangle \tag{11}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)\ \models_s^{l_0}\ s\ \texttt{return}\ e^{l''} \tag{12}$$

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)\ \models_d^{l_0}\ \varepsilon \tag{13}$$

$$\sigma'(\texttt{this})\ \mathcal{V}\ \hat{\zeta}(l_0) \tag{14}$$

where (13) is clear from the definition of $\models_d$. From (C2) we get in particular

$$(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)\ \models_e^l\ r_C^{l_0} \tag{15}$$

$$\forall i \in \{1, \dots, n\}.(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)\ \models_e^l\ v_i^{l_i} \tag{16}$$

$$\forall i \in \{1, \dots, n\}.\hat{\zeta}(l_i) \subseteq \hat{\sigma}(x_i) \tag{17}$$

Note that (17) is universally quantified over all abstract references in $\hat{\zeta}(l_0)$ (the quantification over abstract references is neccessary to perform a method lookup and hence find formal parameters and the label of the return expression). It is justified to apply (17) to our particular x_i, because

$$
\begin{aligned}
(15) \quad &\Rightarrow \quad r_C^{l_0} \; \mathcal{V} \; \hat{\zeta}(l_0) && (18)\\
&\Rightarrow \quad \text{if } \kappa(r) = \varphi_{C'} \text{ then } (C, C') \in \hat{\zeta}(l_0) && (19)
\end{aligned}
$$

Consider definitions 2.7 and 3.6 to convince yourself that (C, C') in (19) is exactly the one used to perform method lookup through $\widehat{call}$, too . Therefore, we may assume (17) for our particular formal parameters x_i. At the same time we have proven (14). It follows from (18) and the fact that $\sigma'(\texttt{this}) = r_C^{l_0}$. We observe that $l_0 \in \gamma(C', m)$ where C' is the type of the object referred to by r_C, which is clear from the definition of the callabel function γ in 3.3. The restriction about private methods in this definition is of no importance here, as we deal with semantic correctness and accessability issues can be dealt with statically as mentioned several times. Among the conditions of theorem 2 we have the correct analysis for a complete program, thus we can derive CLP from fact 3.1 which immediately yields (12). To conclude the proof, we must show (11), where it suffices to consider σ' as κ remains unchanged. Also, we have already proven (14), so that the only thing left to show is

$$\forall i \in \{1, \dots n\}.\sigma'(x_i) \; \mathcal{V} \; \hat{\sigma}(x_i)$$

As for all $i = 1, \dots, n$ $\sigma'(x_i) = v_i^{l_i}$ and $v_i^{l_i} \; \mathcal{V} \; \hat{\zeta}(l_i)$ because of (16), we might apply fact 3.2 together with (17) completing the proof for this case.

$[new]$ From (C1) and (C2) we get

$$
\begin{aligned}
(\texttt{new} \;\; C_0^{l_0}, \langle \sigma, \kappa \rangle) \;\; \rightarrow_\Gamma \;\; (r_{C_0}^{l_0}, \langle \sigma, \kappa[r \mapsto new(C_0)] \rangle) && (20)\\
\{(C_0, C_0)\} \subseteq \hat{\zeta}(l_0) && (21)
\end{aligned}
$$

Showing (R1) means showing $r_{C_0}^{l_0} \; \mathcal{V} \; \hat{\zeta}(l_0)$. This follows immediately from (21) and the definition of $\mathcal{V}$, because $new(C_0) = \varphi_{C_0}$ for some φ, thus r has reference type C_0 and points to an object of type C_0. Again, (R3) follows from (C4) as `this` is not affected in this case. Because of the freshness of r conditioning (20), it suffices to consider the second condition in the definition of $\mathcal{S}$. r_{C_0} cannot occur as value of a variable, because it is fresh. As with this second part, we can as usual restrict ourselves to the case of r_{C_0} as the other references are not affected (freshness!). We have to show $new(C_0)\ \mathcal{O}\ \hat{\kappa}(C_0, C_0)$. This is trivial because *new* initializes all fields with values that always are in $\mathcal{V}$ relation with anything.

Concluding the Proof. As with the cases for which we need to apply the induction hypothesis (IH), most of them are straightforward. We prove them in order of decreasing interest. The most interesting case is about returning from method calls.

$[return2]$ From the assumptions we get the following

$$\begin{array}{rcll}
(C1) & \Rightarrow & \begin{array}{l}(\mathtt{bind}\ \langle\sigma,\kappa\rangle^{l_1}\ \mathtt{in}\ \{\mathtt{return}^{l}\ e^{l'};\}, \langle\sigma_0,\kappa_0\rangle) \\ \quad \rightarrow_{\Gamma}\ (v^{l}, \langle\sigma_0,\kappa'\rangle)\end{array} & (22) \\
\text{because} & & (e^{l_0}, \langle\sigma,\kappa\rangle)\ \rightarrow_{\Gamma}\ (v^{l_0}, \langle\sigma',\kappa'\rangle) & (23) \\
(C2) & \Rightarrow & (\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle)\ \models_s^{l^2}\ \mathtt{bind}\ \langle\sigma,\kappa\rangle^{l_1}\ \mathtt{in}\ \{\mathtt{return}^{l}\ e^{l'};\} & \\
\text{because} & & (\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle)\ \models_s^{l_1}\ \mathtt{return}\ e^{l'} \quad \wedge & (24) \\
 & & \langle\sigma,\kappa\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle \quad \wedge & (25) \\
 & & \sigma(\mathtt{this})\ \mathcal{V}\ \hat{\sigma}(l_1) & (26)
\end{array}$$

In (22) and (23) it is displayed how labels are moved around in the case of returning from method calls. The crucial point is, that the result of the call will be labelled with the original label of the method call as it was added to the bind-construct by $\widehat{call}$. Because of (23), (24) and the definition of the acceptability relation in the case of return-statements, (25) and (26) we may apply (IH) to the transition of (23) which yields

$$\begin{array}{ll}
\langle\sigma',\kappa'\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle & (27) \\
(\hat{\zeta}, \langle\hat{\sigma},\hat{\kappa}\rangle)\ \models_e^{l_1}\ v^{l'} \ \Rightarrow\ v^{l'}\ \mathcal{V}\ \hat{\zeta}(l') & (28)
\end{array}$$

Additionally we have from (C3) and (C4)

$$\begin{array}{rcll}
(C3) & \Rightarrow & \langle\sigma_0,\kappa_0\rangle\ \mathcal{S}\ \langle\hat{\sigma},\hat{\kappa}\rangle & (29) \\
(C4) & \Rightarrow & \sigma_0(\mathtt{this})\ \mathcal{V}\ \hat{\zeta}(l_2) & (30)
\end{array}$$

Now, we have all we need to conclude this case of the proof: (R2) follows from (29), (27) and fact 3.3, because trivially $dom(\kappa_0) \subseteq dom(\kappa')$ – there is no concept of freeing allocated memory in TJ. (R1) follows from (28), fact 3.2, fact 3.4 and fact 3.5, whereas (R3) follows from (30) and fact 3.5. This case is actually one of the most difficult ones, as it requires three auxiliary statements and quite some insight in the definition of the correctness relations.

[$decl3$] From the conditions we get

$$\begin{array}{rcll} (C1) & \Rightarrow & (C\ x = e^{l_0}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ \langle \sigma'[x \mapsto r_C^{l_0}], \kappa' \rangle & \\ \text{because} & & (e^{l_0}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_{C'}^{l_0}, \langle \sigma', \kappa' \rangle) & (31) \\ (C2) & \Rightarrow & (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \ \models_e^l \ e^{l_0} \quad \wedge & (32) \\ & & \forall (C_1, C_2) \in \hat{\zeta}(l_0). \{(C, C_2)\} \subseteq \hat{\sigma}(x) & (33) \end{array}$$

Because of (31), (32), (C3) and (C4) we may apply (IH) yielding

$$\begin{array}{rcll} (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \ \models_e^l \ r_{C'}^{l_0} & \Rightarrow & r_{C'}^{l_0} \ \mathcal{V} \ \hat{\zeta}(l_0) & \\ & \Rightarrow & (C', C'') \in \hat{\zeta}(l_0) & (34) \\ & & \text{where } \kappa'(r) = \varphi_{C''} & \\ \langle \sigma', \kappa' \rangle \ \mathcal{S} \ \langle \hat{\sigma}, \hat{\kappa} \rangle & & & (35) \end{array}$$

There is nothing to show for (R1) and (R3) is equivalent to (C4) as `this` is not affected here. As with (R2) the only thing left to show (because we already have (35)) is the state relation in the case of x, meaning $\sigma'(x) = r_C^{l_0} \ \mathcal{V} \ \hat{\sigma}(x)$. This is equivalent to $(C, C'') \in \hat{\sigma}(x)$, which follows from (33) and (34).

[$cast2$] We get the following from the conditions

$$\begin{array}{rcll} (C1) & \Rightarrow & ((C)e^{l_0\ l_1}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_C^{l_1}, \langle \sigma', \kappa' \rangle) & \\ \text{because} & & (e^{l_0}, \langle \sigma, \kappa \rangle) \ \rightarrow_\Gamma \ (r_{C'}, \langle \sigma', \kappa' \rangle) & (36) \\ (C2) & \Rightarrow & (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \ \models_e^l \ (C)e^{l_0\ l_1} & \\ & \Rightarrow & (\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \ \models_e^l \ e^{l_0} \quad \wedge & (37) \\ & & \forall (C_0, C_1) \in \hat{\zeta}(l_0). \{(C, C_1)\} \subseteq \hat{\zeta}(l_1) & (38) \end{array}$$

Because of (36), (37), (C3) and (C4), we may apply (IH), which immediately yields (R2) and (R3). Furthermore, it yields

$$\begin{array}{rcl} r_{C'} \ \mathcal{V} \ \hat{\zeta}(l_0) & \Rightarrow & (C', C'') \in \hat{\zeta}(l_0) \\ & \text{if} & \kappa'(r) = \varphi_{C''} \\ (38) & \Rightarrow & (C, C'') \in \hat{\zeta}(l_1) \\ & \Rightarrow & r_C \ \mathcal{V} \ \hat{\zeta}(l_1) \end{array}$$

which conludes the proof as it establishes (R1).

As with all the remaining cases, their formal proofs are extremely similar and extremely trivial. This is because they could be replaced with one single rule, if we had introduced semantic contextes. In the condition that justifies the application of such a semantic rule, we always evaluate a subexpression, partially or completely. In the analysis specification, we also get an analysis for the subexpressions – in all so far unproven cases. This will always put us in a position to apply the (IH) which will then immediately yield the desired result. These cases are omitted because of triviality. Only the rule local4 is different, but at the same time pretty much the same as rule ret2, as it deals with exiting a bind-construct.

3.3 Constraint Formulation and Implementation

Now, we want to tackle the problem of how to actually compute a least solution of the control flow analysis described throughout this chapter. In the style of [12, 3.4], whose notation we will apply, we want to reformulate the analysis in terms of set constraints. We shall therefore construct a finite set of constraints called $C_{\models}$, that contains two different types of constraints

$$\begin{array}{rcl} lhs & \subseteq & rhs \\ \{\hat{r}\} \subseteq rhs' & \Rightarrow & lhs \subseteq rhs \end{array}$$

of which the latter shall be read $(\{\hat{r}\} \subseteq rhs' \Rightarrow lhs) \subseteq rhs$ for technical reasons. rhs is of the form $\hat{\zeta}(l)$, $\hat{\sigma}(x)$ or $\hat{\kappa}(\hat{r})f$ while lhs can have the same forms plus the form $\{\hat{r}\}$. The constraints are obtained by expanding the clauses defining $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models (CT, s)$ into $C_{\models}$ like this: For every clause containing a $lhs \subseteq rhs$ condition, the corresponding constraint is included in $C_{\models}$; and for every condition $\forall \hat{r} \in rhs'.lhs \subseteq rhs$ the set

$$\{\ \{\hat{r}\} \subseteq rhs' \Rightarrow lhs \subseteq rhs \mid \hat{r} \in \widehat{\mathbf{Ref}}\ \}$$

of constraints shall be in $C_{\models}$. Note that in contrast to [12] our original specification is more syntax-directed, that is we have all the time restricted ourselves to syntactical constructs appearing in a given program. It is thus clear that we only generate a finite set of constraints and that a solution to $C_{\models}$ also is a solution to $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models (CT, s)$, if we agree on how conditional constraints are interpreted. Again, we use the notation from [12]. It shall hold

$$\begin{array}{l} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)[\![\{\hat{r}\} \subseteq rhs' \Rightarrow lhs]\!] \ = \\ \qquad \begin{cases} (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)[\![lhs]\!] & \text{if } \hat{r} \in (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)[\![rhs']\!] \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

The interpretion of $\hat{\zeta}(l)$, that is $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)[\![\hat{\zeta}(l)]\!]$ is simply $\hat{\zeta}(l)$ again and similarly for the other possible terms appearing in constraints. It is worth noting that in [12] it is distinguished between constraint interpretation and the syntax-directed specification also in terms of notation. As mentioned above, our approach in the syntax-directed specification is already pure syntax, even if formulated as logical clauses. Therefore we do not need to make this distinction. Consult [12] for further details.

Finding a solution to the set of constraints $C_{\models}$ can now be seen as a fixpoint computation over the finite domain $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$. This is a finite complete lattice as been shown in lemma 3.1. If we succeed in defining a monotone function on it, then we get from fixpoint theory, that the least solution to the constraints can be computed by iterating this function. Because of the finiteness of $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ this iteration is guaranteed to terminate. In order to define such a function we need to agree on ls standing for both lhs and $\{\hat{r}\} \subseteq rhs' \Rightarrow lhs$.

$$\begin{array}{rcl} \mathcal{F} & : & \widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}} \to \widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}} \\ \mathcal{F}(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) & = & \big(\mathcal{F}_1(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle), \langle \mathcal{F}_2(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle), \mathcal{F}_3(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \rangle\big) \end{array}$$

where $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ are defined by

$$\begin{array}{rcl} \mathcal{F}_1(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)(l) & = & \bigcup\{(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)[\![ls]\!] \mid ls \subseteq \hat{\zeta}(l) \in C_{\models}\} \\ \mathcal{F}_2(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)(x) & = & \bigcup\{(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)[\![ls]\!] \mid ls \subseteq \hat{\sigma}(x) \in C_{\models}\} \\ \mathcal{F}_3(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)(\hat{r})(f) & = & \bigcup\{(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle)[\![ls]\!] \mid ls \subseteq \hat{\kappa}(\hat{r})(f) \in C_{\models}\} \end{array}$$

It is easy to see that $\mathcal{F}$ indeed defines a monotone function. Consider $(\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1) \sqsubseteq (\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2)$. Due to the definition of $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ as a complete lattice it suffices to show that $\mathcal{F}_i(\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1) \sqsubseteq \mathcal{F}_i(\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2)$ for $i = 1, 2, 3$. This is obvious for ordinary constraints and for conditional constraints it follows from the observation that the lhs' in a conditional constraint $lhs' \subseteq rhs' \Rightarrow lhs \subseteq rhs$ always is a singleton set of an abstract reference $\{\hat{r}\}$ and if $\{\hat{r}\} \subseteq (\hat{\zeta}_1, \langle \hat{\sigma}_1, \hat{\kappa}_1)[\![rhs']\!]$ then also $\{\hat{r}\} \subseteq (\hat{\zeta}_2, \langle \hat{\sigma}_2, \hat{\kappa}_2)[\![rhs']\!]$.

We can compute a least solution to $C_{\models}$ and hence to $(\hat{\zeta}, \langle \hat{\sigma}, \hat{\kappa} \rangle) \models (CT, s)$ by computing the least fixpoint of $\mathcal{F}$. This has indeed been implemented in Standard ML. The source code of the implementation can be found in appendix A. The implementation has been tested on programs of particular interest. We refer to section 5 for examples and applications of the analysis. Finally, a word about complexity. The approach presented here results in a rather poor performance. In [12] it is shown how to obtain a significantly better complexity for solving this problem. The problems we considered in this work, however, could be easily coped with by means of the naive approach described above. Moreover, it is hard to fix a bound on the cost of the algorithm as it depends on many variables: the number of variables, the number of class declarations and the number of labels just to mention some. Of particular interest is the number of field accesses, casts and local declarations, because the clauses for these terms yield a number of conditional constraints that is quadratic in the number of class declarations. This is due to our choice of $\widehat{\mathbf{Ref}}$.

We try to give some approximative bound. We shall denote the number of expressions in a program with E. This includes subexpressions, too. C shall stand for the number of class declarations. $O(E)$ should be a bound on the sum of the number of labels, variables, local declarations, field accesses and so on, while $O(C^2)$ obviously bounds the number of abstract references. If we only had field accesses and casts, thus generating only conditional constraints, we had an $O(EC^2)$ bound on the number of constraints. This is expected to be significantly smaller in practice, though. We can view a solution $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)$ as an $O(L+X+F*C^2)$ tuple of values in $\widehat{\mathbf{Value}}$, where L is the number of labels, X the number of variables and F the number of fields. Certainly, this is again bound by $O(EC^2)$. Since $\widehat{\mathbf{Value}}$ is a lattice of height $O(C)$, $\widehat{\mathbf{Cache}} \times \widehat{\mathbf{State}}$ has height $O(EC^3)$ and hence we can compute a fixpoint of $\mathcal{F}$ starting at the bottom element in at most $O(EC^3)$ iteration. Our approach needs to consider all $O(EC^2)$ constraints to determine the value of each of the $O(EC^2)$ components of the iterant; this yields an overall $O(E^3C^7)$ bound on the cost. Once again, this approximation is not very exact as it does not take into account variable names and labels among others.

4 A Multi-Threaded Extension

In this section, we want to extend TJ with some concurrency primitives. A formal semantics shall be given, which makes use of so-called *Event Spaces* as introduced in [3]. Then we shall show how to exploit the analysis from section 3 to perform control flow analyses on concurrent programs. A correctness proof for the extended analysis is based on exhibiting a bisimulation between the original TJ semantics and the extended semantics in case of a single thread.

4.1 The TJ_c Language

We shall now define *Concurrent TJ*, TJ_c, as a multi-threaded extension of TJ. TJ_c will only be equipped with very few concurrency primitives. Among these, there are multiple threads of execution running in parallel and synchronization via shared objects. The semantics shall be like in Java. Consider the Java Language Specifaction [10, chapter 17] for a detailed description of the concurrency model of Java. Compared to the full Java concurrency model, we most notably lack synchronization via wait sets and notificaton, dynamic thread creation as well as starting and stopping of threads. These restrictions are because of the design of our semantics and analysis. In the respective sections, we will be able to give some proper arguments in favour of these restrictions. But let us now define TJ_c

Definition 4.1 (TJ_c) *The abstract syntax of TJ_c is defined by the grammar in figure 4 where the clause for programs is replaced with*

$$Program_c \ ::= \ ClassDecls \quad \theta_1 : Stms \ \ldots \ \theta_n : Stms$$

where $n \geq 1$ is a natural number and the clause for statments is extended by

$$Stms \ ::= \ \ldots \ | \ \texttt{synchronized}(Expr) \ Stms$$

We shall define **ThreadId** $= \{\theta_1, \ldots, \theta_n\}$ *to be a set of thread names and θ ranging over it. The θ_i shall be distinct. Other metavariable conventions and syntactic categories are like in the definition of TJ.*

The meaning of definition 4.1 is as follows: The θ_i shall be distinct names of threads of execution. Upon the start of the execution of such a program, all statements corresponding to the respective threads shall be executed in parallel obeying some rules to be defined. The number of threads is static and a constant. We do not allow dynamic thread creation. Synchronization happens via shared objects. The expression in a synchronized statement shall evaluate to a reference to an object. If some thread wants to execute the statements inside a synchronized-statement, it must acquire a lock on the object referred to by the expression. This means, that no other thread may perform write operations on this object any more. After having executed the synchronized code, a thread releases its lock again, making the object accesible for other threads. A more formal definition of this informal description is given in the part on semantics.

4.2 Semantics

The semantics of TJ_c is parametric in the notion of *event spaces*. These have been introduced by Cenciarelli, Knapp, Reus and Wirsing in [3] and [4]. They correspond roughly to configurations in Winskel's event structures [14]. We will follow the respective part of [3] almost literally in our presentation of event spaces. The difference is in the underlying semantics. Ours is somewhat larger than the one in [3] with respect to the sequential part, which is about the same as presented in [4]. The latter work, however, comprises many more concurrency features. To include all of them in our semantics and, even worse, in the analysis, would result in a much too complex and combinatorial description. We shall rather concentrate on some essential concurrent features and view them as an extension to TJ rather than a whole new language. This is very much in the style of [3], which we will closely follow throughout this semantics part. Even if not in quotation marks, the part on event spaces is taken literally from this work – modulo some adaptions in notation and the underlying semantics. The latter one leads to some larger differences in the second part of the presentation of the semantics.

4.2.1 Event Spaces

The execution of a Java program comprises many threads of computation runnning in parallel. Threads exchange information by operating on values and objects residing in a shared main memory. As explained in the Java Language Specification [10], each thread also has a private working memory in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of each variable. These are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. Moreover, there are rules which regulate the locking and unlocking of objects, by means of which threads synchronize with each other. These rules are given in [10, chapter 17] and formalized in this section as well-formedness conditions for structures called event spaces. In the next section event spaces are included in the configurations of multi-threaded TJ to constrain the applicability of certain operational rules. In accord with [10], the terms *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock* and *Unlock* are used here to name actions which describe the activity of the memories during the execution. *Use* and *Assign* denote the above mentioned actions on the private working memory. *Read* and *Load* are used for a loosely coupled copying of data from the main memory to a working memory and dually *Store* and *Write* are used for copying data from a working memory to the main memory.

After this informal description, we are prepared to formally define event spaces. We use the following conventions on metavariables. We let A range over any of the 8 actions described above, B over the set of thread actions and C over the set of memory actions, that is:

$$\begin{array}{rcl} B & \in & \{Use, Assign, Load, Store, Lock, Unlock\} \\ C & \in & \{Read, Write, Lock, Unlock\} \end{array}$$

Definition 4.2 (Event Space) *Let us write* $\widetilde{\mathbf{Adr}}$ *for the domain* $\mathbf{Adr} \times \mathbf{FieldId} \times \mathbf{ClassId}$ *and let* $\tilde{r}$ *range over its elements. Elements of each of the following domains shall be called* **actions**.

$$\begin{array}{rcl} (A, \theta, \tilde{r}, v) & \in & \{Assign, Store, Read\} \times \mathbf{ThreadId} \times \widetilde{\mathbf{Adr}} \times \mathbf{Value} \\ (A, \theta, \tilde{r}) & \in & \{Use, Load, Write\} \times \mathbf{ThreadId} \times \widetilde{\mathbf{Adr}} \\ (A, \theta, o) & \in & \{Lock, Unlock\} \times \mathbf{ThreadId} \times \mathbf{Object} \end{array}$$

Events *are instances of actions, thought of as happening at different times during execution. An* **event space** *is a poset of events, in which every chain can be enumerated monotonically with respect to the arithmetical ordering* $0 \leq 1 \leq 2 \leq \ldots$ *of natural numbers; and which satisfies the conditions (1-15) below.*

We shall use the same tuple notation for actions and their instances and let lower case letters stand for either. Sometimes we omit components of an action or event: we may write $(Read, \tilde{r})$ for $(Read, \theta, \tilde{r}, v)$ when θ and v are not relevant. A triple (A, θ, x) is read "θ performs an A action on x", where $x \equiv \tilde{r}$ or $x \equiv o$, while a quadruple $(A, \theta, \tilde{r}, v)$ is read "θ performs an A action on $\tilde{r}$ with value v". The (A, θ, x) notation shall from now on stand for both the second and third clause in the definition of events. The ordering in the poset of events shall be thought of as chronological ordering during execution. Note that we deviate from [3] in the notion of extended adresses $\tilde{r}$. These terms are called left-values in [3] and denote adresses for every single field of an object, whereas our model of the heap only comprises one adress per object, through which we afterwards access single fields of this object. As we on top of that also handle the case of shadowing fields, we need to inlcude both a field and a class name in our notion of left-values or extended adresses $\tilde{r}$. The 15 conditions mentioned in the defintion formalize directly the rules of [10, chapter 17] and are expressed by clauses of the form

$$\forall a \in \eta.(\Phi \Rightarrow ((\exists b_1 \in \eta.\Psi_1) \vee \ldots \vee (\exists b_n \in \eta.\Psi_n)))$$

where a and b_i are lists of events, η is an event space and $\forall a \in \eta.\Phi$ means that Φ holds for all tuples of events in η matching the elements of a (and similarly for $\exists b_i \in \eta.\Psi_i$). Such statements are abbreviated by adopting the following conventions: quantification over a is left implicit, when all events in a appear in Φ; quantification ober b_i is left implicit, when all events in b_i appear in Ψ_i. Moreover, a rule of the form $\forall a \in \eta.(true \Rightarrow \ldots)$ is written $a \Rightarrow (\ldots)$. We include some short, informal explanation of the rules and refer to [10] for more detail.

The actions performed by any one thread are totally ordered, and so are the actions performed by the main memory for any one variable [10, 17.2, 17.5].

$$(B,\theta),(B',\theta') \;\Rightarrow\; (B,\theta) \leq (B',\theta') \;\vee\; (B',\theta') \leq (B,\theta) \qquad (1)$$

$$(C,x),(C',x') \;\Rightarrow\; (C,x) \leq (C',x') \;\vee\; (C',x') \leq (C,c) \qquad (2)$$

Hence the occurences of any action (A,θ,x) are totally ordered in an event space. The term $(A,\theta,x)_n$ denotes the n-th occurence of (A,θ,x) in a given event space, if such an event exists, and is undefined otherwise. When two indices m and n are applied in a rule to instances of the same action, it is meant that $m \neq n$.

A *Store* action by θ on $\tilde{r}$ must intervene between an *Assign* by θ of $\tilde{r}$ and a subsequent *Load* by θ of $\tilde{r}$. Less formally, a thread is not permitted to lose its most recent assign [10, 17.3]:

$$\begin{array}{l}(Assign,\theta,\tilde{r}) \leq (Load,\theta,\tilde{r}) \Rightarrow \\ \qquad (Assign,\theta,\tilde{r}) \leq (Store,\theta,\tilde{r}) \leq (Load,\theta,\tilde{r})\end{array} \qquad (3)$$

A thread is not permitted to write data from its working memory back to main memory for no reason [10, 17.3]:

$$\begin{array}{l}(Store,\theta,\tilde{r})_m \leq (Store,\theta,\tilde{r})_n \Rightarrow \\ \qquad (Store,\theta,\tilde{r})_m \leq (Assign,\theta,\tilde{r})_n \leq (Store,\theta,\tilde{r})_n\end{array} \qquad (4)$$

Threads start with an empty working memory and new variables are created only in main memory and not initially in any thread's working memory [10, 17.3]:

$$\begin{aligned}(Use, \theta, \tilde{r}) &\Rightarrow (Assign, \theta, \tilde{r}) \leq (Use, \theta, \tilde{r}) \ \vee \\ &\quad\ (Load, \theta, \tilde{r}) \leq (Use, \theta, \tilde{r}) \qquad (5)\\ (Store, \theta, \tilde{r}) &\Rightarrow (Assign, \theta, \tilde{r}) \leq (Store, \theta, \tilde{r}) \qquad (6)\end{aligned}$$

A *Store* action transmits the contents of the thread's working copy of a variable to main memory [10, 17.3]:

$$\begin{aligned}&(Assign, \theta, \tilde{r}, v)_n \leq (Store, \theta, \tilde{r}, v') \Rightarrow \\ &\quad v = v' \ \vee \ (Assign, \theta, \tilde{r}, v)_n \leq (Assign, \theta, \tilde{r})_m \leq (Store, \theta, \tilde{r}, v') \qquad (7)\end{aligned}$$

Each *Load* and *Write* action is uniquely paired respectively with a matching *Read* or *Store* action that preceeds it [10, 17.2, 17.3]:

$$\begin{aligned}(Load, \theta, \tilde{r})_n &\Rightarrow (Read, \theta, \tilde{r})_n \leq (Load, \theta, \tilde{r})_n \qquad (8)\\ (Write, \theta, \tilde{r})_n &\Rightarrow (Store, \theta, \tilde{r})_n \leq (Write, \theta, \tilde{r})_n \qquad (9)\end{aligned}$$

The actions on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested [10, 17.3]:

$$(Store, \theta, \tilde{r})_m \leq (Load, \theta, \tilde{r})_n \Rightarrow (Write, \theta, \tilde{r})_m \leq (Read, \theta, \tilde{r})_n \qquad (10)$$

A thread is not permitted to unlock a lock that it does not own. Only one thread at a time is permitted to lay claim to a lock, and moreover a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed [10, 17.5]:

$$\begin{aligned}&(Unlock, \theta, o)_n \Rightarrow (Lock, \theta, o)_n \leq (Unlock, \theta, o)_n \qquad (11)\\ &(Lock, \theta, o)_n \leq (Lock, \theta', o) \ \wedge \ \theta \neq \theta' \Rightarrow \\ &\quad (Unlock, \theta, o)_n \leq (Lock, \theta', o) \qquad (12)\end{aligned}$$

If a thread is to perform an unlock action on any lock, it must first copy all assigned values in its working memory back out to main memory [10, 17.6]:

$$\begin{aligned}&(Assign, \theta, \tilde{r}) \leq (Unlock, \theta) \Rightarrow \\ &\quad (Assign, \theta, \tilde{r}) \leq (Store, \theta, \tilde{r})_n \leq (Write, \theta, \tilde{r})_n \leq (Unlock, \theta) \qquad (13)\end{aligned}$$

A lock action acts as if it flushes all variables from the thread's working memory; before use they must be assigned or loaded from main memory [10, 17.6]:

$$
\begin{aligned}
&(Lock, \theta) \leq (Use, \theta, \tilde{r}) \Rightarrow \\
&\quad (Lock, \theta) \leq (Assign, \theta, \tilde{r}) \leq (Use, \theta, \tilde{r}) \vee \\
&\quad (Lock, \theta) \leq (Read, \theta, \tilde{r})_n \leq (Load, \theta, \tilde{r})_n \leq (Use, \theta, \tilde{r})
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
&(Lock, \theta) \leq (Store, \theta, \tilde{r}) \Rightarrow \\
&\quad (Lock, \theta) \leq (Assign, \theta, \tilde{r}) \leq (Store, \theta, \tilde{r})
\end{aligned} \tag{15}
$$

[3] argues that each of the above rules correponds to one rule in [10] and conversely, any rule in [10] not included above can be derived in this axiomatization.

Now we are left to define how an event can be added to an existing event space. Like in [3] we shall use the $\oplus$ operator for adjoining an event to an event space. Moreover, the same operator shall be used in case of adjoining a sequence of events. Formally

Definition 4.3 ($\oplus$ Operator) *A new event $a = (A, \theta, x)$ is adjoined to an events space η by extending the excution order as follows: if A is a thread action, then $b \leq a$ for all instances of b of (B, θ) in η; if A is a main memory action, then $c \leq a$ for all instances c of $(Read, \theta, \tilde{r})$ in η, and if A is $Write$ then then $c \leq a$ for all instances c of $(Store, \theta, \tilde{r})$ in η. The term $\eta \oplus a$ denotes the event space thus obtained, provided it obeys the above rules, and it is otherwise undefined. If η is an event space and $a = (a_1, a_2, \ldots, a_n)$ a sequence of events, we shall write $\eta \oplus a$ for $\eta \oplus a_1 \oplus a_2 \oplus \ldots \oplus a_n$.*

Event spaces can now be used to record historical information of the computation which constrains the excution of certain actions according to the language specification, and hence the applicability of certain operational rules. In general, this notion of event spaces modelling the Java concurrency model can be added to almost any semantics of Java or sublanguages that want to capture the concurrent behaviour of Java. On the other hand, choosing a different axiomatization and different event spaces together with the same operational rules, yields different multi-threaded semantics as shown in [4].

4.2.2 Extended Rules

We shall now see how event spaces fit together with our operational rules from section 2. To do so we need to extend our semantic domains, too. Domains that are not introduced here and used below shall be identical to the ones defined in 2.1. Basically, the only changes concern states, configurations and the semantic relation.

Definition 4.4 (Extended Domains) *Let ES be the domain of event spaces and let η range over it. t shall range over single-threaded TJ terms.*

$$
\begin{array}{rcll}
\langle\tilde{\sigma},\kappa\rangle & \in & \widetilde{\mathbf{State}} & := (\mathbf{ThreadId} \to \mathbf{Id} \to \mathbf{Value}) \times \\
 & & & (\mathbf{Adr} \to \mathbf{Object}) \\
T & \in & TJ_c\,Term & := (\mathbf{ThreadId} \times TJTerm \;\cup\; \mathbf{ThreadId})^\star \\
 & & \widetilde{\mathbf{Config}} & := TJ_c\,Term \times \widetilde{\mathbf{State}} \times ES
\end{array}
$$

In addition to that, let $\theta_1 : t_1 \mid \ldots \mid \theta_n : t_n$ *range over* $TJ_c\,Term$*, too.*

Note that we have introduced some inconsistency as with respect to metavariable conventions. We let t range over single-threaded TJ terms here, whereas it was used to range over types before. However, it will always be clear from the context which one is meant. The most notable difference to the well-known domains from section 2 is the notion of multiple states. Each thread identifier is mapped to its own stack part of the state, whereas the heap remains unchanged. This is because we need a common heap for synchronization between the threads. As before, **ThreadId** is finite and includes all thread identifiers appearing in a program. Event spaces are now a part of the configuration and progress in the semantics depends strongly on them as we shall see in the operational rules.

Definition 4.5 (Extended Relation) *Let* $\to_c \subseteq \widetilde{\mathbf{Config}} \times \widetilde{\mathbf{Config}}$ *denote the extended semantic relation, which is defined by the operational rules in table 13. A primed rule in table 13 shall replace the unprimed rule of the single-threaded semantics and the other rules extend the single-threaded semantics. All the rules from the single-threaded semantics that are not replaced with a primed version in table 13 shall be replaced with a rule that is obtained by applying the following convention: Replace every transition of the form* $(t, \langle\sigma,\kappa\rangle) \;\to_\Gamma\; (t', \langle\sigma',\kappa'\rangle)$ *with a transition of the form*

$$\theta : t, \langle\tilde{\sigma},\kappa\rangle, \eta \;\to_c\; \theta : t', \langle\tilde{\sigma}',\kappa'\rangle, \eta'$$

where $\tilde{\sigma}(\theta) = \sigma$ *and* $\tilde{\sigma}' = \tilde{\sigma}[\theta \mapsto \sigma']$.

In the rules, we make use of the functions val and $store$. $val(\eta,\theta,\tilde{r})$ denotes the latest value of $\tilde{r}$ assigned or loaded by θ in η and $store(\eta,\theta,\tilde{r})$ is written for the oldest unwritten value of $\tilde{r}$ stored by θ in η: More formally: let an event $(Store,\theta,\tilde{r})_n$ in η be called unwritten if $(Write,\theta,\tilde{r})_n$ is undefined in η; then, $store(\eta,\theta,\tilde{r}) = v$ if there exists an unwritten $(Store,\theta,\tilde{r},v)_n$ such

$$[\mathrm{id}'] \quad \frac{}{\theta : x, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : \tilde{\sigma}\ \theta\ x, \langle \tilde{\sigma}, \kappa \rangle, \eta}$$

$$[\mathrm{access2}'] \quad \frac{}{\begin{array}{l}\theta : r_C.f, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : val(\eta, \theta, (r, f, C)), \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Use, \theta, (r, f, C))\end{array}}$$

$$[\mathrm{assign4}'] \quad \frac{}{\begin{array}{l}\theta : r_C.f = v, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : , \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Assign, \theta, (r, f, C), v)\end{array}}$$

$$[\mathrm{par}] \quad \frac{\theta : t, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : t', \langle \tilde{\sigma}', \kappa' \rangle, \eta'}{\theta : t \mid T, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : t' \mid T, \langle \tilde{\sigma}', \kappa' \rangle, \eta}$$

$$[\mathrm{synchro1}] \quad \frac{\theta : e, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : e', \langle \tilde{\sigma}', \kappa' \rangle, \eta'}{\begin{array}{l}\theta : \texttt{synchronized}(e)s, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : \texttt{synchronized}(e')s, \langle \tilde{\sigma}', \kappa' \rangle, \eta'\end{array}}$$

$$[\mathrm{synchro2}] \quad \frac{\theta : s, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : s', \langle \tilde{\sigma}', \kappa' \rangle, \eta'}{\begin{array}{l}\theta : \texttt{synchronized}(v)s, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : \texttt{synchronized}(v)s', \langle \tilde{\sigma}', \kappa' \rangle, \eta'\end{array}}$$

$$[\mathrm{lock}] \quad \frac{\theta : e, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ \theta : r_C, \langle \tilde{\sigma}', \kappa' \rangle, \eta'}{\begin{array}{l}\theta : \texttt{synchronized}(e)s, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : \texttt{synchronized}(r_C)s, \langle \tilde{\sigma}', \kappa' \rangle, \eta' \oplus (Lock, \theta, \kappa'(r))\end{array}}$$

$$[\mathrm{unlock}] \quad \frac{}{\begin{array}{l}\theta : \texttt{synchronized}(r_C), \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad \theta : , \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Unlock, \theta, \kappa(r))\end{array}}$$

$$[\mathrm{read}] \quad \frac{}{\begin{array}{l}T, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad T, \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Read, \theta, (r, f, C), \kappa\ r\ (f, C))\end{array}}$$

$$[\mathrm{load}] \quad \frac{}{T, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ T, \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Load, \theta, \tilde{r})}$$

$$[\mathrm{store}] \quad \frac{}{T, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \ T, \langle \tilde{\sigma}, \kappa \rangle, \eta \oplus (Store, \theta, \tilde{r}, v)}$$

$$[\mathrm{write}] \quad \frac{}{\begin{array}{l}T, \langle \tilde{\sigma}, \kappa \rangle, \eta \ \rightarrow_c \\ \quad T, \langle \tilde{\sigma}, \kappa[r \mapsto store(\eta, \theta, (r, f, C))] \rangle, \eta \oplus (Write, \theta, (r, f, C))\end{array}}$$

Table 13: Extended Operational Rules

that for any unwritten $(Store, \theta, \tilde{r})_m$ we have $n \leq m$. Moreover, we write $\kappa\ r\ (f, C)$ as an abbreviation for $\varphi(f, C)$ where $\kappa(r) = \varphi_{C'}$.

Further commments on the rules involve the requirement that the expression evaluated in a synchronized-statement must evaluate to a reference. In [3], the last four rules in table 13 are called *silent actions*, as they do not perform any computation but add something to the event space only. Note that the update on the heap that happened in rule assign4 in the single-threaded semantics is now delayed. It will happen by means of a silent $Write$ action (rule write), which is guaranteed by condition (13) in the definition of event spaces. All the rules only apply if a $\oplus$ operation in the rule yields a valid event space. Thus we can impose the Java concurrency model on our TJ semantics.

4.3 Control Flow Analysis

Due to complexity and the design of [3], on which we based our presentation of a multi-threaded semantics, we picked a concurrency model with rather restricted means of synchronization. Now we shall reason about the statically determined – in contrast to dynamically created – number of threads. First of all, we need to define extended abstract domains, then we have to extend the acceptability relation of section 3 by the new constructs and finally we will argue about correctness.

The extended abstract domains differ only slightly from the original ones, that is the only change deals with abstract values. The new domain for abstract values looks like this:

$$(\hat{v}, \theta) \in \widehat{\mathbf{Value}}_c \ := \ \mathcal{P}(\mathbf{ThreadId} \times \mathcal{P}(\widehat{\mathbf{Ref}}))$$

All the other extended domains are obtained by substituting $\widehat{\mathbf{Value}}_c$ for $\widehat{\mathbf{Value}}$. We shall use the same metavariables ranging over the extended domains as before.

Intuitevely: Not only do we track for each label, variable or field which kind of objects they can refer to, but we trace threads at the same time. For each label, variable or field we determine a superset of the following set: the set of pairs made up of a thread that can reach this point during its execution and the set of abstract values that might appear at this point during execution of the thread. Formally, we have to extend the acceptability relation by the following clauses:

$$
\begin{array}{rl}
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l & \texttt{synchronized}(e)s \\
\iff & (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_e^l e \quad \wedge \quad (\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_s^l s
\end{array}
$$

$$
\begin{array}{rl}
(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_\theta & CT \quad \theta_i : s_i \ \ldots \ \theta_n : s_n \\
& \quad \forall i \in \{1, \ldots, n\}. \\
\iff & \left\{ \begin{array}{l}
(\hat{\zeta}_i, \langle\hat{\sigma}_i, \hat{\kappa}_i\rangle \models (CT, s_i) \quad \wedge \\
\forall l \in \mathbf{Label}.(\theta_i, \hat{\zeta}_i(l)) \in \hat{\zeta}(l) \quad \wedge \\
\forall x \in \mathbf{Id}.(\theta_i, \hat{\sigma}_i(x)) \in \hat{\sigma}(x) \quad \wedge \\
\forall \hat{r} \in \widehat{\mathbf{Ref}} \forall f \in \mathbf{FieldId}.(\theta_i, \hat{\kappa}_i \ \hat{r} \ f) \in \hat{\kappa} \ \hat{r} \ f
\end{array} \right.
\end{array}
$$

This specification implies immediately that a multi-threaded analysis can be performed by analyzing every single thread alone. Hence we can even use the same implementation as in the single-threaded case. As we are merely interested in different kinds (i.e. class type and reference type) of objects and as threads are objects in Java, we see that there can only be a constant number of different types of threads in a Java program. Dynamically created threads can only assume a constant number of types and we do not distinguish threads of the same type. This means that due to the design of our analysis we would not be able to distinguish between almost all dynamically created threads, if we considered them objects. That is why we restrict ourselves to a constant number.

What about correctness? We can make use of a result stated in [3] – called *conservativity* there. [3] exhibits a bisimulation between its single-threaded semantics and the semantics obtained by adding event spaces and considering only one thread. Although this proof bases on the semantics presented there, the most interesting cases of the proof involve the silent actions mentioned above and no particular operational rule. Even if not formally carried out, this leads to some confidence that such a result could be easily shown for our semantics, too. Once we have such a result as well as the correctness result for the single-threaded case proven in section 3, we can easily deduce some kind of correctness for our multi-threaded analysis. As the analysis does not consider synchronization, it is obvious that analyzing every single thread alone yields a superset of the actual traces of threads, because synchronization could only restrict the control flow for a single thread.

5 Applications

Formally defining an analysis, proving it semantically correct and even extending it with multiple threads is fun. Still, someone could ask why to do so. This section tries to give an answer by demonstrating how to make use of the analysis. In the first part, we will use our implementation for the single-threaded case to analyze small examples, which demonstrate typical useful features of the analysis. In the second part, we will become more theoretical again and impose a Bell-LaPadula security model on the multi-threaded semantics from section 4, which will lead to a notion of secure programs. Finally, we will show that our analysis is able to prove a program secure in terms of the imposed Bell-LaPadula model.

5.1 The Single-Threaded Case

In both upcoming examples, we will proceed as follows: First, the example is given as TJ source code, then we show some of the constraints generated and finally the solution is presented. Both constraints and solution have been generated by the implementation, which allows to output TEX source files.

5.1.1 Example 1

Here is the first example to analyze:

$$
\begin{array}{l}
\texttt{class A extends Object \{\}} \\
\\
\texttt{class B extends A \{} \\
\quad \texttt{public int f;} \\
\quad \texttt{public void m() \{ this.f = 1}^{0}\texttt{; return; \}} \\
\texttt{\}} \\
\\
\texttt{A a = new A()}^{1}\texttt{;} \\
\texttt{((B)a}^{2}\texttt{).m()}^{4}\texttt{;}
\end{array}
$$

And here are four of the generated constraints. In fact, these are the constraints that really contribute to finding the solution. Playing with the implementation will reveal that there are many more constraints generated. This is mostly because of the number of abstract references.

$$
\begin{array}{rcl}
\{(A,A)\} & \subseteq & \hat{\zeta}(1) \\
\{(A,A)\} \subseteq \hat{\zeta}(1) & \Rightarrow & \{(A,A)\} \subseteq \hat{\sigma}(a) \\
\hat{\sigma}(a) & \subseteq & \hat{\zeta}(2) \\
\{(A,A)\} \subseteq \hat{\zeta}(2) & \Rightarrow & (B,A) \subseteq \hat{\zeta}(3)
\end{array}
$$

The constraints are listed in the order that they shall be evaluated and in exactly the same way control flows in this tiny program. Note that the second constraint was generated by the rule for local declarations and is hence conditional. The same applies to the last one, that was generated by the rule for casts. If this example is adjusted to Java syntax, then it will compile. However, there is an obviously illegal cast, because a is a pure A object, that cannot be casted down to B. This is revealed by our analysis. So here is the solution, where we leave out the $\hat{\kappa}$ component, because nothing happens with fields here. So this is the $\hat{\zeta}$ component

$\hat{\zeta}$	0	1	2	3	4
	$\{\}$	$\{(A, A)\}$	$\{(A, A)\}$	$\{(B, A)\}$	$\{\}$

and here is $\hat{\sigma}$

$\hat{\sigma}$	a
	$\{(A, A)\}$

The analysis shows that (B, A) might occur at label 3. Every occurence of (B, A) must be illegal because there cannot be a B reference pointing to an A object: B is a subclass of A. In this case the analysis – used in the back-end of a compiler – could issue an illegal-cast warning. This is the first purpose our analysis can serve: *detecting illegal casts*. As we only compute supersets, we cannot for sure predict illegal casts, but still issue warnings.

5.1.2 Example 2

Consider the somewhat larger example 2:

```
class C extends Object {}

class D extends C {}

class A extends Object {
  public C f;
}

class B extends A {
  public D f;
  public D m() { this.f = new D()^0; return this.f^1; }
}

A ab = new B()^2;
C cd = ((B)ab^3)^4.m()^5;
if (ab^6 instanceof D)^7 then ab.f = new C()^8 else skip;
```

and some chosen constraints generated at analyzing it

$$
\begin{array}{rcll}
\{(B,B)\} & \subseteq & \hat{\zeta}(2) & (1)\\
\{(B,B)\} \subseteq \hat{\zeta}(2) & \Rightarrow & (A,B) \subseteq \hat{\sigma}(ab) & (2)\\
\hat{\sigma}(ab) & \subseteq & \hat{\zeta}(3) & (3)\\
\{(A,B)\} \subseteq \hat{\zeta}(3) & \Rightarrow & \{(B,B)\} \subseteq \hat{\zeta}(4) & (4)\\
\{(D,D)\} & \subseteq & \hat{\zeta}(0) & (5)\\
\{(B,B)\} \subseteq \hat{\zeta}(4) & \Rightarrow & \hat{\zeta}(0) \subseteq \hat{\kappa}((B,B),f) & (6)\\
\{(B,B)\} \subseteq \hat{\zeta}(4) & \Rightarrow & \hat{\kappa}((B,B),f) \subseteq \hat{\zeta}(1) & (7)\\
\{(B,B)\} \subseteq \hat{\zeta}(4) & \Rightarrow & \hat{\zeta}(1) \subseteq \hat{\zeta}(5) & (8)\\
\{(D,D)\} \subseteq \hat{\zeta}(5) & \Rightarrow & \{(C,D)\} \subseteq \hat{\sigma}(cd) & (9)\\
\hat{\sigma}(cd) & \subseteq & \hat{\zeta}(6) & (10)\\
\{(C,C)\} & \subseteq & \hat{\zeta}(8) & (11)\\
\{(A,B)\} \subseteq \hat{\sigma}(ab) & \Rightarrow & \hat{\zeta}(8) \subseteq \hat{\kappa}((A,B),f) & (12)
\end{array}
$$

Note that we only picked a few chosen constraints contributing immediately to finding the solution. In this example, 111 constraints are actually generated. This blow-up results from having 4 class declarations, three of which are empty. Therefore we generate lots of unneccessary constraints, as we have a big number of abstract references. Remember: the complexity of the algorithm depends heavily on the number of class declarations. The smaller the classes the larger is their influence on the number of constraints relatively to the size of the actual code.

In fact, the constraints in the above ordering trace exactly the control flow in the program: (1) updates label 2 at the new-expression. Due to the local declaration rule we include (A, B) in the set of abstract values for the variable ab. Then we pass this value on to label 3 applying the rule for how to handle variable expressions. As we know by now, the condition in (4) is satisfied, thus we include (B, B) in label 4, according to the rule about casts. Now, the interesting part comes, the method call. The callabel function γ will return 4 as the only callabel for method m. It is actually the only method call. Because of that, $\hat{\zeta}(4)$ will represent the `this` reference in the method body. (5) is clear again, while the conditions of (6-8) are true as we know by now, because the `this` reference actually has abstract value (B, B), which followed from (4). (6) and (7) are trivial value passing inside the method body, whereas (8) is the constraint for passing the return value back to the label of the whole method call, i.e. label 5. (9) is yet another local declaration and (10) passes a value from a variable to its annotated label. Here, the flow breaks, because we do not use our knowledge so far to predict the outcome of the instanceof-expression. Nothing is passed to label 7, because boolean values are not considered in this analysis. This is object to some further extensions. (11) and (12) are trivial again.

By now, it should be clear how the solution looks like. Again, we do not include all the information about $\hat{\kappa}$, as it is non-empty only at two places.

$\hat{\zeta}$	0	1	2	3	4	5
	$\{(D, D)\}$	$\{(D, D)\}$	$\{(B, B)\}$	$\{(A, B)\}$	$\{(B, B)\}$	$\{(D, D)\}$

$\hat{\zeta}$	6	7	8
	$\{(C, D)\}$		$\{(C, C)\}$

$\hat{\sigma}$	ab	cd
	$\{(A, B)\}$	$\{(C, D)\}$

$\hat{\kappa}$	f
(B,B)	$\{(D, D)\}$
(A,B)	$\{(C, C)\}$

As seen, this example exhibits many more features of the analysis, like return value passing or the concept of analyzing method bodies with respect to `this` references. We can also see, how the concept of shadowing fields is realized in the analysis. In the $\hat{\kappa}$ component of the solution, we see two different fields, that – on the semantics side – actually belong to the same object, namely the one created by `new B()`. Its A field contains (C, C) and its B field (D, D). How can we use this information? Again, we shall see application possibilities in compiler back-ends. We can e.g. predict that the instanceof-expression will evaluate to true, because the only abstract value in the test expression is actually an instance of D (see $\hat{\zeta}(6)$). We could in fact transform the if-statement into the then-part only, as we know the outcome of `instanceof`. So there are applications in *program transformations and optimizations*, too. Other optimisations based on our analysis might be method inlining. If we can predict, that at a certain method call, only one kind of object occurs this could be done. This means that we can predict some *dynamic dispatches.* One could definitely think of even more ways of using the rich information gained from the analysis.

5.2 The Multi-Threaded Case

The applications presented so far have basically dealt with enabling program transformations or predicting runtime behaviour of single-threaded TJ. Not only do we switch to TJ_c now, but also do we consider a completely different application of our analysis. We want to make use of our correctness result from section 3, in order to prove a program secure. Security is defined by means of a Bell-LaPadula security model (BLP) as introduced in section 1.3. Remember the definitions of section 1.3, because we will need them now. We shall define a BLP on the basis of the multi-threaded TJ_c semantics before using our analysis in order to prove this model secure.

Remember that we need to say, what subjects, objects, access rights and security levels are. These make up the basis of a BLP. Then we can proceed defining a state set, where states shall be understood as configurations of a program. In order to introduce security levels, we need some further annotations of programs. Let L be a partially ordered set of security levels, that must be specified by a user, and write λ for the elements of L. Then we stipulate the following annotations

$$\begin{array}{rcl} Prog_c & ::= & ClassDecls \;\; \theta_1^{\lambda_{\theta_1}} : s_1 \;\ldots\; \theta_n^{\lambda_{\theta_n}} : s_n \\ ClassDecls & ::= & \texttt{class } C^{\lambda_C} \texttt{ extends } \ldots \end{array}$$

Every thread identifier must be annotated with a security level and so must be the class names. Assume an arbitrary but from now on fixed and annotated program, **ThreadId** the finite set of thread names defined in the program and **Object** the semantic domain of objects for this given program. We can then define

- **ThreadId** to be the set of subjects in the security model
- **Object** to be the set of objects in the security model

- $A = \{execute, append, read, write\}$ the set of access operations – see section 1.3.

- $L = \{\lambda_C, \lambda_\theta \mid C \in \mathbf{ClassId}, \theta \in \mathbf{ThreadId}\}$ the partially ordered set of security levels with ordering $\leq$ specified by the user

- $\mathcal{M} = (\mathbf{ThreadId} \times \mathbf{ClassId}) \to \mathcal{P}(A)$ the domain of access matrices, that is to be defined by the user. Deviating from the requirement of an access matrix to be a mapping from subjects and objects to access rights, we replaced objects with class names. This is justified, because all objects of the same type C have the same security level λ_C. From now on, we assume one fixed element $M \in \mathcal{M}$.

- $f_S : \mathbf{ThreadId} \to L$ where $f_S(\theta) := \lambda_\theta$ the function mapping threads to their annotated security level. f_C shall be defined equivalently, as there is no way of changing security levels of subjects in our model. $f_O : \mathbf{Object} \to L$ is defined by $f_O(\varphi_C) := \lambda_C$, i.e. the security level of an object is the same as the security level of its type. Obviously, (f_S, f_C, f_O) can be assumed fixed, too. We shall write f for this triple and $\mathcal{F}$ for the domain made up of such triples

All these sets are defined relative to one fixed program, that determines the domains. As states in our model correspond to configurations of the TJ_c semantics under evaluation, i.e. elements of $\widetilde{\mathbf{Config}}$, we need to define a function mapping configurations to state sets, where a state set is defined by

$$(b, M, f) \in B \times \mathcal{M} \times \mathcal{F}$$

M and f are arbitrary but fixed in our context. They are in particular not dependent on configurations, which are mapped to B by the two functions:

$$\begin{aligned} C &: \widetilde{\mathbf{Config}} \to B \\ c &: \mathbf{ThreadId} \times Term \times (\mathbf{Adr} \to \mathbf{Object}) \to B \end{aligned}$$

where

$$\begin{aligned} C(\theta_1 : t_1 \ \ldots \ \theta_n : t_n, \langle\sigma, \kappa\rangle, \eta) &:= \{c(\theta_i, t_i, \kappa) \mid i = 1, \ldots, n\} && (1) \\ c(\theta, r_C.f := v, \kappa) &:= (\theta, \kappa(r), append) && (2) \\ c(\theta, r_C.f, \kappa) &:= (\theta, \kappa(r), read) && (3) \\ c(\theta, r_C.m(v_1, \ldots, v_n), \kappa) &:= (\theta, \kappa(r), write) && (4) \end{aligned}$$

c shall be undefined otherwise. As usual, we only make the union over all elements for which c is defined in (1). (2-4) mirror directly the three ways of accessing objects in our model. We do not have any *execute* access in our model, as there not really is a configuration, where it makes sense to speak of this kind of access. The choices are deliberate, but seem most meaningful in our setting. In (2), we assign to the field of an object, hence alter it, but obviously we do not observe anything of the object. (3) is exactly the other way round and (4), i.e. method call, can both observe and alter an object. In all other configurations, it does not actually make sense to speak of object access, therefore c shall be undefined there. If we furthermore agree on adding our fixed M and f to the result of C, then we have a true mapping from configurations to the states of the BLP model.

Once we have this model, we can state the three properties, that need to be checked for proving a state, i.e. a configuration, secure. These are the ss-, the $\star$- and the ds-property – see section 1.3. A configuration is secure if and only if C applied to it (plus M and f) are secure in the BLP model just defined. A program is secure, if all the configurations it might assume during computation are secure. Of course, checking this for a program, i.e. for every configuration, is unfeasible, as we cannot compute all configurations statically. Now theorem 2 comes in very handy, as we can make some provably safe predictions about all configurations with the help of our analysis. Checking the security of every configuration can then be reduced to checking the solution $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle)$ of our analysis, which is certainly feasible. Before stating the theorem, we have to introduce some further notation, though, in order to make the formulation of the theorem more elegant.

Definition 5.1 *Be P a TJ_c program and $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_\theta$ an analysis result. The sets $A, R, W \subseteq \widehat{\mathbf{Value}}_c$ of append, read and write value-sets are defined by*

- *If $X \subseteq \mathbf{Id}$ is the set of identifiers x through which a field is accessed and assigned, i.e. $x.f = v$ is in P; and if L_{Cm} is the set of all labels $l \in \gamma(C, m)$ in case a field assignment is made through* `this` *in method m of class C, i.e.* ***this****$.f = v$ occurs in m of C, then*

$$\begin{aligned} A \ := \ & \{(\theta, \hat{v}) \in \hat{\sigma}(x) \mid x \in X\} \cup \\ & \{(\theta, \hat{v}) \in \hat{\zeta}(l) \mid l \in L_{Cm}, C \in \mathbf{ClassId}, m \in \mathbf{MethId}\} \end{aligned}$$

  where γ is the callabel function defined in 2.7.

- *If L_f is the set of all labels that label a field access, i.e var.f^l is in P for some field f, then*

$$R \ := \ \{(\theta, \hat{v}) \in \zeta(l) \mid l \in L_f\}$$

- *If L_m is the set of labels that label the receiver expression of a method call, i.e. $e^l.m(\ldots)$ is in P for some method m, then*

$$W \ := \ \{(\theta, \hat{v}) \in \zeta(l) \mid l \in L_m\}$$

We can now state the result about program security.

Theorem 3 (Program Security) *Let P be a TJ_c program and assume a correct analysis, i.e. $(\hat{\zeta}, \langle\hat{\sigma}, \hat{\kappa}\rangle) \models_\theta P$. Then P is secure if the following holds*

- $\forall(\theta, \hat{v}) \in R \cup W \forall(C, C') \in \hat{v}.\lambda_{C'} \leq f_S(\theta)$
- $\forall(\theta, \hat{v}) \in A \cup W \forall(C, C') \in \hat{v}.f_C(S) \leq \lambda_{C'}$
- *For all $l \in$ **Label** and for all $x \in$ **Id** it holds*

$$\begin{array}{l} \forall(\theta, \hat{v}) \in A \forall(C, C') \in \hat{v}.append \in M(\theta, \lambda_{C'}) \\ \forall(\theta, \hat{v}) \in R \forall(C, C') \in \hat{v}.read \in M(\theta, \lambda_{C'}) \\ \forall(\theta, \hat{v}) \in W \forall(C, C') \in \hat{v}.write \in M(\theta, \lambda_{C'}) \end{array}$$

Note that we can only prove the "if" part of this theorem. This is because of the nature of our analysis. It yields a superset of what can happen at certain program points. If this is still secure, then so is the program. In contrast, there can be situations, where the analysis reveals some flaws, whereas these might never occur in a real computation. Because of this imprecision, we cannot claim "only if" in theorem 3. Moreover, the three security conditions of theorem 3 mirror directly the three properties of secure states in BLP – just in terms of our analysis. In the definition of the $\star$-property there is another requirement, which will always be met in the context of configurations. It is about one thread accessing several objects at the same time. These situations will not occur in our notion of BLP, which is clear from the definition of C.

What we gain is that the conditions as described in theorem 3 can be easily checked, once we have obtained an analysis. *Security guarantees* with respect to a BLP security model is thus maybe the most interesting application of our analysis.

Eventually, a word about correctness: A correctness proof for this theorem is trivial, as we can make use of the semantic correctness theorem. It says in particular that throughout the computation, i.e. in every possible configuration, the actual state of the configuration is in $\mathcal{S}$ relation with the abstract state of our solution. This means that for every actual value, that either a variable can assume or that can occur at certain program points, its abstract description is included in our analysis result for this particular variable or program point. As security levels of objects are captured by these abstract descriptions, the abstract state and cache suffice to prove security for the whole execution of a program.

6 Concluding Remarks

This section consists of a loose coupling of comments, suggestions, shortcomings, possibilities for improvements, future work and former versions of this work. Furthermore, its contribution is discussed.

We shall begin with a word about former versions of this work. In the original version we employed a natural operational semantics rather than a structural one. The decision in favour of the structural verision was made, when it stood clear that concurrency should be dealt with, too. Certainly, speaking about concurrency in terms of operational semantics requires the structural version. Also, in the beginning, we intended only to analyze a small calculus-like object-orientated language. During the work on this, we discovered that the techniques and ideas just as well could be applied to a somewhat larger language and so TJ appeared gradually.

Further work could of course include further extensions of TJ. There are two possiblities: on the multi-threaded side, one could base the semantics on a more sophisticated concurrency model, like the one described in [4]. The latter work has e.g. incorportated wait and notification sets as another means of synchronization, which would represent a natural extensions of TJ. On the other side, namely the thread-independent one, one could think of adding concepts like exception handling, constructors, method overloading or protected fields and methods to TJ. None of these should pose any conceptual difficulties but merely technical ones. We are confident that this could even happen without any changes to the existing semantics but simply as an add-on. This means that the semantics and analysis presented here are rather modular with this respect.

One particular comment has to be made on the realization of method invocation and local variable declaration in the structural operational semantics. We have chosen to introduce the bind-construct as a piece of syntax. Another alternative is encoding the growing and shrinkink stack in the semantic domains rather than in the syntax. This would result in a leaner syntax, but on the other hand it would complicate the handling of the domains, too. Moreover, we could have chosen to introduce two different bind-constructs for either method calls or local declarations. The benefit would consist in a clearer and less subtle notation. The bind-construct behaves e.g. differently depending on whether it is exited via a return or not. Again, something else would become more complicated, namely the semantic rules and the analysis specification. Still, two bind-constructs seem like an attractive improvement and have mainly not been included, because the work was almost finished when the idea came up.

Then we have to mention possible improvements to the analysis, both theoretically and practically. As with theory, it seems attractive to make use of static analyses in order to improve the analysis. This affects the call-label funtion in particular. One could easily think of mean examples (with a lot of overwritten methods), where the call-label function as presented here would perform really badly, because it does not take anything into acount but method names. Information from static analysis could help improving it tremendously. This improvement has not been made as we wanted to concentrate on dynamic behaviour rather than static analysis. Consider the three works cited in section 1.1. for how static analyses for TJ-like languages can be performed. Also, one could add data-flow analysis to the pure control flow analysis here. Such an analysis would not abstract away from basic values but collect them, too. By doing so, one might e.g. predict the outcome of evaluating the condition of an if-statement, thus finding out that some code will never be excuted and thus giving space to improve the control-flow part again. How this can be done is described in [12]. It is not really complicated. [12] describes an extension with contextes, too, what will result in a k-CFA rather than in a 0-CFA as presented here. All these features could be easily added to the analysis as it is.

The given implementation is very naive and very close to the abstract specification. Ways of improving it are e.g. revealed in [12] by means of a graph-based formulation of the fixpoint computation. Another option is to write an interface to high performing constraint solving tools. These improvements are of no interest to this work, though, but could be subject to some practical projects, because the properties of the analysis would not be affected by changing the implementation. Our implementation shall rather prove, that even the most naive implementation is able to analyze some programs. Moreover, it reveals enough about the ideas and concepts behind the analysis.

Eventually, we shall conclude with what we consider the major contributions of this work. We have formally defined a structural operational semantics for a substantial subset of Java, which even includes concurrency primitives. Based on this, we have developped a powerful control flow analysis for this rather large language and we succeeded in proving crucial properties: the existence of solutions and semantic correctness. The power of this approach to control flow analysis can be appreciated, if one considers the size of the language and the naive implementation, which, however, produces usable results. These results can be used in various contexts of program optimisation and transformation, compilers and, most notably, security. It has been shown, that the famous Bell-LaPadula security model can be imposed on the execution of a multi-threaded program and that the analysis is powerful enough to statically prove its security with respect to this model.

References

[1] David E. Bell and Leonard LaPadula. *Secure Computer System: Unified Exposition and Multics Interpretation*. In ESD-TR-75-306, ESD-AFSC, Hanscom AFB

[2] Chiara Bodei, Pierpaolo Degano, Flemming Nielson and Hanne Riis Nielson. *Control Flow Analysis for the pi-calculus*. In Proceedings CONCUR 1998, Lecture Notes in Computer Science 1466

[3] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus and Martin Wirsing. *From sequential to Multi-Threaded Java - An Event-Based Structural Operational Semantics*. In Lecture Notes in Computer Science 1349

[4] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus and Martin Wirsing. *An Event-Based Structural Operational Semantics of Multi-Threaded Java*. In Lecture Notes in Computer Science 1523

[5] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990

[6] Sophia Drossopoulou and Susan Eisenbach. *Java is Type-Safe – Probably*. In Proceedings ECOOP 1997, Lecture Notes in Computer Science 1241.

[7] Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen. *Classes and Mixins*. In Proceedings POPL 1998

[8] Kirsten Lackner Solberg Gasser, Flemming Nielson and Hanne Riis Nielson. *Systematic Realisation of Control Flow Analysis for CML*. In Proceedings ICFP 1997

[9] Dieter Gollmann. *Computer Security*. Wiley, 1999

[10] James Gosling, Bill Joy and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading 1996

[11] Atsushi Igarashi, Benjamin Pierce and Philip Wadler. *Featherweight Java*. In Proceedings OOPSLA 1999

[12] Flemming Nielson, Hanne Riis Nielson and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin 1999

[13] Gordon Plotkin. *A Structural Approach to Operation Semantics*. Technical Report DAIMI FN-19, Aarhus University, 1981

[14] Glynn Winskel. *An Introduction to Event Structures.* In Lecture Notes in Computer Science 354

A Source Code

The source code presented here is available upon request to the authour.

A.1 A Set Datatype

In the implementation, we make use of a set datatype, which is included in every distribution of Standard ML. Here, we shall only give the signature of this datatype and its operations. Remember to include the respective files, when you want to use the code presented here. Include the following files:

```
lib-base-sig.sml
lib-base.sml
ord-key-sig.sml
ord-set-sig.sml
binary-set-fn.sml
```

from path `../src/smlnj-lib/Util`, where the dots represent the installation directory of your SML distribution. Here are the signatures that are important for understanding our source code.

```
signature ORD_KEY =
  sig
    type ord_key

    val compare : ord_key * ord_key -> order

  end (* ORD_KEY *)


signature ORD_SET =
  sig

    structure Key : ORD_KEY

    type item = Key.ord_key
    type set

    val empty : set
(* The empty set *)

    val singleton : item -> set
(* Create a singleton set *)

    val add  : set * item -> set
    val add' : (item * set) -> set
(* Insert an item. *)

    val addList : set * item list -> set
(* Insert items from list. *)

    val delete : set * item -> set
(* Remove an item. Raise NotFound if not found. *)
```

```
    val member : set * item -> bool
(* Return true if and only if item is an element in the set *)

    val isEmpty : set -> bool
(* Return true if and only if the set is empty *)

    val equal : (set * set) -> bool
(* Return true if and only if the two sets are equal *)

    val compare : (set * set) -> order
(* does a lexical comparison of two sets *)

    val isSubset : (set * set) -> bool
(* Return true if and only if the first set is a subset of the second *)

    val numItems : set ->  int
(* Return the number of items in the table *)

    val listItems : set -> item list
(* Return an ordered list of the items in the set *)

    val union : set * set -> set
        (* Union *)

    val intersection : set * set -> set
        (* Intersection *)

    val difference : set * set -> set
        (* Difference *)

    val map : (item -> item) -> set -> set
(* Create a new set by applying a map function to the elements
 * of the set.
         *)

    val app : (item -> unit) -> set -> unit
(* Apply a function to the entries of the set
         * in decreasing order
         *)

    val foldl : (item * 'b -> 'b) -> 'b -> set -> 'b
(* Apply a folding function to the entries of the set
         * in increasing order
         *)

    val foldr : (item * 'b -> 'b) -> 'b -> set -> 'b
(* Apply a folding function to the entries of the set
         * in decreasing order
         *)

    val filter : (item -> bool) -> set -> set

    val exists : (item -> bool) -> set -> bool

    val find : (item -> bool) -> set -> item option

  end (* ORD_SET *)
```

A.2 The Analysis Code

The code consists of two parts, a signature and an implementation part. The implementation is very close two the way, we presented the analysis. It should be understandable after having read the analysis part and with the help of the comments. Here is the datatypes we use and the signature of the analysis.

```
type Label = int
type ClassId = string
type FieldId = string
type MethId = string
type Id = string

datatype Program = PROG of ClassDecl list * Stms
and ClassDecl    = CLASSDECL of ClassId * ClassId * FieldDecl list * MethDecl list
and FieldDecl    = FIELDDECL of AccMod * Type * FieldId
and MethDecl     = METHDECL of AccMod * Type * MethId * Arg list * Stms * ReturnExp
and ReturnExp    = RETURN | RETURNE of Exp
and Arg          = ARG of Type * Id
and AccMod       = PRIVATE of ClassId | PUBLIC
and Type         = VOID | BOOL | INT | CT of ClassId
and Stms         = EXPRSTM of Exp | IFTHENELSE of Exp * Stms * Stms
 | LOCAL of LocalDecl list * Stms | SEQ of Stms * Stms
 | SKIP | ASSIGN of Var * Exp
and LocalDecl    = LOCALDECL of Type * Id * Exp
and Exp          = E of Expr * Label
and Expr         = Z of int | B of bool | NULL | BINOP of string * Exp * Exp
 | UNOP of string * Exp | CAST of ClassId * Exp | VAR of Var
 | NEW of ClassId | INSTANCEOF of Exp * ClassId
 | CALL of Exp * MethId * Exp list
and Var          = ID of Id | THIS | ACCESS of Var * FieldId


signature Analysis =

sig

structure RefSet : ORD_SET
structure CSet : ORD_SET
type Ref = ClassId * ClassId

type Value = RefSet.set
type Cache = Label -> Value
type Object = FieldId -> Value
type State = (Id -> Value) * (Ref -> Object)

type Term
type Constraint

val generateConstraints : Program -> CSet.set
val solve : Program -> Cache * State
val constraints : CSet.set
val solution : Cache * State

val pp_constraints : string -> unit
val pp_solution : string -> unit

end
```

The main idea is writing a functor that takes as argument a program, its labels, variable and field names. The functor then performs the analysis and makes it available via two pretty-print functions that generate TEXsource code. This source code can be used in some mathematical environments with only slight adaptions.

```
functor CFA (val P : Program
             val ids : Id list
             val labels : Label list
             val fields : FieldId list) :> Analysis =

   struct

(************************************************)
(* types and structures required by the signature *)
(************************************************)

        type Ref = ClassId * ClassId
        structure RefSet = BinarySetFn (
           struct
                      type ord_key = Ref
                fun compare ((c1,c2),(d1,d2)) =
                     let val comp1 = String.compare (c1,d1)
                        val comp2 = String.compare (c2,d2)
                    in if comp1 = EQUAL then comp2 else comp1
                    end
        end )
        type Value = RefSet.set
        type Cache = Label -> Value
        type Object = FieldId -> Value
        type State = (Id -> Value) * (Ref -> Object)

        datatype Term = ZETA of Label
                      | SIGMA of Id
                            | KAPPA of Ref * FieldId
                           | REF of Ref

        datatype Constraint = N of Term * Term
                            | C of Ref * Term * Term * Term

        structure CSet = BinarySetFn (
             struct
                    type ord_key = Constraint
                     fun compare_r ((c1,c2):Ref,(d1,d2):Ref) =
                   let val comp1 = String.compare (c1,d1)
                       val comp2 = String.compare (c2,d2)
                   in if comp1 = EQUAL then comp2 else comp1
                   end
                     fun compare_t (ZETA _, SIGMA _) = GREATER
                 | compare_t (ZETA _, KAPPA _) = GREATER
                 | compare_t (ZETA _, REF _) = GREATER
                 | compare_t (ZETA l1, ZETA l2) = Int.compare(l1,l2)
                 | compare_t (SIGMA _, ZETA _) = LESS
                 | compare_t (SIGMA x, SIGMA y) = String.compare(x,y)
                 | compare_t (SIGMA x, KAPPA _) = GREATER
                 | compare_t (SIGMA _, REF _) = GREATER
                 | compare_t (KAPPA _, ZETA _) = LESS
                 | compare_t (KAPPA _, SIGMA _) = LESS
                 | compare_t (KAPPA _, REF _) = GREATER
                 | compare_t (KAPPA (r1,f1), KAPPA (r2,f2)) =
```

```
                 let val c1 = compare_r (r1,r2)
                     val c2 = String.compare (f1,f2)
                 in if c1 = EQUAL then c2 else c1
                 end
              | compare_t (REF _,KAPPA _) = LESS
              | compare_t (REF _,SIGMA _)  = LESS
              | compare_t (REF _,ZETA _) = LESS
              | compare_t (REF r1,REF r2) = compare_r(r1,r2)
          fun compare (N (t1,t2), N (t1',t2')) =
                let val c1 = compare_t (t1,t1')
                    val c2 = compare_t (t2,t2')
                in if c1 = EQUAL then c2 else c1
                 end
               | compare (N _, C _) = LESS
             | compare (C _, N _) = GREATER
             | compare (C(r,t1,t2,t3),C(r',t1',t2',t3')) =
                     let val c0 = compare_r(r,r')
                     val c1 = compare_t (t1,t1')
                    val c2 = compare_t (t2,t2')
                    val c3 = compare_t (t3,t3')
                 in if c0 = EQUAL
                    then if c1 = EQUAL
                        then if c2 = EQUAL then c3 else c2
                        else c1
                   else c0
                    end
        end )


(****************************************)
(* Creating the class environment Gamma  *)
(****************************************)

        type Gamma =  ClassId ->
                      ClassId * (FieldId -> AccMod * Type) *
                            (MethId -> AccMod * Type * Arg list * Stms * ReturnExp)


        fun GammaGen (PROG(cdl,s)) =
               foldl (fn (CLASSDECL(c1,c2,fdl,mdl),f) =>
                      (fn c => if c=c1 then (c2,GammaGenF fdl,GammaGenM mdl)
                                       else f c))
                  (fn c => ("",fn f => (PUBLIC,VOID) ,
                               fn m => (PUBLIC,VOID,[],SKIP,RETURN))) cdl

        and GammaGenF fdl =
                foldl (fn (FIELDDECL(acm,t,fid),f) =>
                      (fn ff => if ff=fid then (acm,t) else f fid))
                (fn f => (PUBLIC,VOID)) fdl

        and GammaGenM mdl =
                foldl (fn (METHDECL(acm,t,mid,argl,s,rexp),f) =>
                      (fn m => if m=mid then (acm,t,argl,s,rexp) else f mid))
            (fn m => (PUBLIC,VOID,[],SKIP,RETURN)) mdl
        (* no option type used for undefined methods *)

(*********************************)
(* The call-label function, \gamma *)
(*********************************)

        structure LabelSet = BinarySetFn (
              struct
```

```
                type ord_key = Label
               val compare = Int.compare
      end )

   fun call (PROG(cdl,s),c,m) =
       let val gamma = GammaGen (PROG(cdl,s))
        in        if #1((#3(gamma c))m) = PRIVATE c
              then LabelSet.union (collect_s(m,#4((#3(gamma c))m)),
                         collect_re(m,#5((#3(gamma c))m)))
             else LabelSet.union((collect_p (m,cdl)),(collect_s (m,s)))
       end

   and collect_p (m,cdl) = foldl (fn (c,lset) =>
        LabelSet.union(collect_c(m,c),lset)) (LabelSet.empty) cdl

   and collect_c (m,CLASSDECL(_,_,_,mdl)) =
       foldl (fn (me,lset) => LabelSet.union(collect_m(m,me),lset))
       (LabelSet.empty) mdl

   and collect_m (m,METHDECL(_,_,_,_,s,e)) =
       LabelSet.union (collect_s(m,s),collect_re(m,e))

   and collect_s (m,s) = case s of
       EXPRSTM e => collect_e(m,e)
     | IFTHENELSE (e,s1,s2) =>
          LabelSet.union(LabelSet.union(collect_s(m,s1),collect_s(m,s2)),
                         collect_e(m,e))
     | LOCAL (ldl,s) => LabelSet.union(collect_s(m,s),collect_l(m,ldl))
     | SEQ (s1,s2) => LabelSet.union(collect_s(m,s1),collect_s(m,s2))
     | ASSIGN (x,e) => collect_e(m,e)
     | SKIP => LabelSet.empty

   and collect_l (m,ldl) =
       foldl (fn (LOCALDECL(_,_,e),lset) => LabelSet.union(collect_e(m,e),lset))
      (LabelSet.empty) ldl

   and collect_re (m,RETURNE e) = collect_e(m,e)
     | collect_re (m,RETURN) = LabelSet.empty

   and collect_e (m,E(e,l)) = case e of
       BINOP(_,e1,e2) => LabelSet.union(collect_e(m,e1),collect_e(m,e2))
     | UNOP (_,e1) => collect_e(m,e1)
     | CAST (_,e1) => collect_e(m,e1)
     | INSTANCEOF (e1,_) => collect_e(m,e1)
     | CALL (E(e1,l1),m1,el) =>
          LabelSet.union (LabelSet.union(
             if m1=m then LabelSet.singleton(l1) else LabelSet.empty,
             collect_e(m,E(e1,l1))),
             foldl (fn (e,lset) => LabelSet.union(collect_e(m,e),lset))
             (LabelSet.empty) el)
     | _ => LabelSet.empty


(************************************************)
(* computing all abstract references, that can *)
(* occur in a compiled Java File               *)
(************************************************)

   val comp = fn gamma : Gamma =>
           let fun issubclass("Object",c2) = "Object"=c2
             | issubclass(c1,c2) = if (c1=c2) orelse ((#1(gamma c1))=c2) then true
```

```
                                    else (issubclass((#1(gamma c1)),c2))
            in fn (c1,c2) => issubclass(c1,c2)
            end

        val refs = fn (PROG(cdl,s),gamma:Gamma) =>
               let val classes = foldl (fn (CLASSDECL(c,_,_,_),l) => c::l) ["Object"] cdl
                 val sc = comp gamma
           in List.filter (fn (c1,c2) => sc(c1,c2) orelse sc(c2,c1))
                       (List.concat (map (fn c1 => map (fn c2 => (c1,c2)) classes) classes))
           end

        val refs_p = refs (P : Program, GammaGen P)

(*************************)
(* Constraint Generation *)
(*************************)

val generateConstraints = fn P:Program =>
    let val gamma = GammaGen P
        val call' = fn (c,m) => call (P,c,m)
        val absrefs = refs (P,gamma)
    in let

       fun ConsGen (PROG (cdl,s)) = CSet.union
                                  (foldl (fn (CLASSDECL(c,_,_,mdl) ,cset)
                                           => CSet.union(cons_md(c,mdl),cset)) (CSet.empty) cdl,
                                   cons_s(~1,s))

       and cons_md (c,mdl) =
           foldl
           (fn (METHDECL(_,_,m,_,s,e) ,cset) =>
               let val llist = LabelSet.listItems (call' (c,m))
               in CSet.union (cset,
                  foldl (fn (l,cset') => CSet.union(cset', CSet.union(cons_s(l,s),cons_re(l,e))))
                  (CSet.empty) llist)
               end)
           (CSet.empty) mdl

       and cons_re (l,RETURN) = CSet.empty
         | cons_re (l,RETURNE e) = cons_e(l,e)

       and cons_s (l,s) = case s of
           EXPRSTM e => cons_e(l,e)
         | IFTHENELSE (e,s1,s2) => CSet.union(CSet.union(cons_s(l,s1),cons_s(l,s2)),cons_e(l,e))
         | LOCAL (ldl,s) => CSet.union(cons_d(l,ldl), cons_s(l,s))
         | SEQ (s1,s2) => CSet.union(cons_s(l,s1),cons_s(l,s2))
         | SKIP => CSet.empty
         | ASSIGN (v,E(e,l0)) => CSet.union (cons_e(l,E(e,l0)),
              case v of
                 ID x => CSet.singleton (N(ZETA l0, SIGMA x))
               | THIS => CSet.singleton (N(ZETA l0, ZETA l))
               | ACCESS (ID x, f) => foldl
                        (fn (r,cset) => CSet.union(cset, CSet.singleton (C(r,SIGMA x,ZETA l0,KAPPA(r,f)))))
                        (CSet.empty) absrefs
               | ACCESS (THIS, f) => foldl
                        (fn (r,cset) => CSet.union(cset, CSet.singleton (C(r,ZETA l,ZETA l0,KAPPA(r,f)))))
                        (CSet.empty) absrefs
               | ACCESS _ => CSet.empty (* only one field dereferncing *)
              )

       and cons_d (l,ldl) =
           foldl (fn (LOCALDECL(CT T,x,E(e,l0)),cset) =>
```

```
                  CSet.union(cons_e(l,E(e,l0)),CSet.union(cset,
                  foldl (fn ((c1,c2),cset') =>
                  CSet.union(cset',CSet.singleton (C((c1,c2),ZETA l0,REF(T,c2),SIGMA x))))
                  (CSet.empty) absrefs))

                | (LOCALDECL(_,x,e),cset) => CSet.union(cset,cons_e(l,e))  )
          (CSet.empty) ldl

      and cons_e (l,E(e',l0)) = case e' of
          BINOP(_,e1,e2) => CSet.union(cons_e(l,e1),cons_e(l,e2))
        | UNOP(_,e1) => cons_e(l,e1)
        | CAST(c,E(e1,l1)) => CSet.union(cons_e(l,E(e1,l1)), foldl
             (fn ((c1,c2),cset) => CSet.union(cset,CSet.singleton(C((c1,c2),ZETA l1, REF(c,c2), ZETA l0))))
             (CSet.empty) absrefs )
        | VAR v => (case v of
              ID x => CSet.singleton(N(SIGMA x, ZETA l0))
            | THIS => CSet.singleton(N(ZETA l, ZETA l0))
            | ACCESS(ID x,f) => foldl (fn (r,cset) =>
                          CSet.union(cset,CSet.singleton(C(r,SIGMA x, KAPPA(r,f), ZETA l0))))
                          (CSet.empty) absrefs
            | ACCESS(THIS,f) => foldl (fn (r,cset) =>
                          CSet.union(cset,CSet.singleton(C(r,ZETA l, KAPPA(r,f), ZETA l0))))
                          (CSet.empty) absrefs
            | ACCESS _ => CSet.empty)
        | NEW c => CSet.singleton(N(REF(c,c),ZETA l0))
        | INSTANCEOF (e,c) => cons_e(l,e)
        | CALL(E(e,l1),m,el) => CSet.union(cons_e(l,E(e,l1)),
             let fun mlookup(c,m) =
                     let val me = (#3(gamma c)) m
                     in if (me = (PUBLIC,VOID,[],SKIP,RETURN)) andalso (c <> "Object")
                        then mlookup(#1(gamma c),m)
                        else me
                     end
             in foldl
                ( fn ((c1,c2),cset) =>
                     let val(acm,t,parlist,_,rexp) = mlookup(c2,m)
                     in CSet.union(cset, CSet.union(
                       (case (rexp,t) of
                           (RETURNE(E(_,l2)),CT(_)) => CSet.singleton(C((c1,c2),ZETA l1, ZETA l2, ZETA l0))
                         | _ => CSet.empty),
                       foldl (fn ((E(_,l3),ARG(t,x)),cset) =>
                          CSet.union(cset,
                          case t of CT _ => CSet.singleton(C((c1,c2),ZETA l1, ZETA l3, SIGMA x))
                                  | _ => CSet.empty )
                       )
                       (CSet.empty) (ListPair.zip(el,parlist))))
                     end
                )
                (CSet.empty) absrefs
             end)
        | _ => CSet.empty

      in ConsGen P
      end
   end


(*********************)
(* Constraint Solving *)
(*********************)

fun eval ((cache,(sigma,kappa)) : Cache * State ,ZETA l) = cache l
```

```
  | eval ((_,(sigma,_)), SIGMA x) = sigma x
  | eval ((_,(_,kappa)), KAPPA(r,f)) = kappa r f
  | eval (_, REF r) = RefSet.singleton r

fun eval' (solution : Cache * State, r, t1, t2) =
    if RefSet.member(eval(solution,t1),r)
    then eval(solution,t2) else RefSet.empty

fun equals ((cache,(sigma,kappa)) : Cache * State,(cache',(sigma',kappa')) : Cache * State) =
    (foldl (fn (x,t) => (RefSet.equal(sigma x, sigma' x)) andalso t) true ids) andalso
    (foldl (fn (l,t) => (RefSet.equal(cache l, cache' l)) andalso t) true labels) andalso
    let val k = (map (fn r => kappa r) refs_p)
        val k' = (map (fn r => kappa' r) refs_p)
    in ListPair.foldl (fn (kf,kf',t) =>
                          (foldl (fn (f,tt) =>
                          (RefSet.equal(kf f, kf' f)) andalso tt) true fields)
                          andalso t) true (k,k')
    end

val fix' = fn cset : CSet.set =>
    let val f1 = fn (cache,(sigma,kappa)) : Cache * State =>
            fn l : Label => let val new = CSet.filter (fn (N(_,ZETA l')) => l=l'
                                                    | (C(_,_,_,ZETA l')) => l=l'
                                                    | _ => false) cset
                            in CSet.foldl (fn (N(t,_ ),rset) =>
                                    RefSet.union(rset,eval((cache,(sigma,kappa)),t))
                                            | (C(r,t1,t2,_),rset) =>
                                    RefSet.union(rset,eval'((cache,(sigma,kappa)),r,t1,t2))
                                           ) (RefSet.empty) new
                            end

        val f2 = fn (cache,(sigma,kappa)) : Cache * State =>
            fn x : Id => let val new = CSet.filter (fn (N(_, SIGMA x')) => x=x'
                                                 | (C(_,_,_,SIGMA x')) => x=x'
                                                 | _ => false) cset
                         in CSet.foldl (fn (N(t,_ ),rset) =>
                                           RefSet.union(rset,eval((cache,(sigma,kappa)),t))
                                         | (C(r,t1,t2,_),rset) =>
                                           RefSet.union(rset,eval'((cache,(sigma,kappa)),r,t1,t2))
                                           ) (RefSet.empty) new
                         end

        val f3 = fn (cache,(sigma,kappa)) : Cache * State =>
            fn r : Ref => fn f : FieldId =>
               let val new = CSet.filter (fn (N(_, KAPPA(r',f'))) => r=r' andalso f=f'
                                           | (C(_,_,_,KAPPA(r',f'))) => r=r' andalso f=f'
                                           | _ => false) cset
               in CSet.foldl (fn (N(t,_ ),rset) =>
                                    RefSet.union(rset,eval((cache,(sigma,kappa)),t))
                               | (C(r,t1,t2,_),rset) =>
                                    RefSet.union(rset,eval'((cache,(sigma,kappa)),r,t1,t2))
                                           ) (RefSet.empty) new
               end

    in fn (cache,(sigma,kappa)) : Cache * State =>
         (f1 (cache,(sigma,kappa)), ( f2 (cache,(sigma,kappa)), f3 (cache,(sigma,kappa)) ) )
    end

fun solve (P:Program) =
    let val cset = generateConstraints P
        val fix = fix' cset
        fun iter (solution : Cache * State) =
```

```
            let val new = fix solution
            in if equals(solution,new)
               then solution
               else iter new
            end
    in iter (fn _ => RefSet.empty, (fn _ => RefSet.empty, fn _ => fn _ => RefSet.empty))
    end

(***************)
(* The Results *)
(***************)

val constraints = generateConstraints P
val solution = solve P

(*******************)
(* Pretty Printing *)
(*******************)

val pp_constraints = fn name =>
    let fun pp_r (c1,c2) = "("^c1^","^c2^")"
        fun pp_t (SIGMA x) = "\\hat\\sigma("^x^")"
          | pp_t (ZETA l)  = "\\hat\\zeta("^(Int.toString l)^")"
          | pp_t (KAPPA (r,f)) = "\\hat\\kappa("^(pp_r r)^","^f^")"
          | pp_t (REF r) = pp_r r
        fun pp_c (N(t1,t2)) = ""^(pp_t t1)^"&\\subseteq&"^(pp_t t2)^"\\\\\n"
          | pp_c (C(r,t1,t2,t3)) = (pp_r r)^"\\ \\subseteq\\ "^(pp_t t1)^
                                   "&\\Rightarrow&"^(pp_t t2)^
                                   "\\ \\subseteq\\ "^(pp_t t3)^"\\\\\n"
    in let val stream = TextIO.openOut (name^".tex")
       in TextIO.output (stream,
           (foldl (fn (c,s) => (pp_c c)^s) ("") (CSet.listItems constraints)))
          before TextIO.closeOut stream
       end
    end


val pp_solution = fn name =>
    let fun pp_val v = if RefSet.isEmpty v then "" else
              "\\lbrace\\ "^
              (RefSet.foldl (fn ((c1,c2),s) => "("^c1^","^c2^")") " " v)^
              "\\rbrace"

        fun pp_zeta z =
           "\\begin{tabular}{|"^(foldr (fn (_,s) => "c|"^s) "" labels)^"}\n"^
           "\\hat\\zeta"^(foldr (fn (l,s) => "&"^(Int.toString l)^s) "" labels)^
           "\\\\\\hline\n"^
           (foldr (fn (l,s) => "&"^(pp_val (z l))^s) "" labels)^
           "\n\\end{tabular}"

        fun pp_sigma si =
           "\\begin{tabular}{|"^(foldr (fn (_,s) => "c|"^s) "" ids)^"}\n"^
           "\\hat\\sigma"^(foldr (fn (x,s) => "&"^x^s) "" ids)^
           "\\\\\\hline\n"^
           (foldr (fn (x,s) => "&"^(pp_val (si x))^s) "" ids)^
           "\n\\end{tabular}"

        fun pp_kappa k =
           "\\begin{tabular}{|"^(foldr (fn (_,s) => "c|"^s) "" fields)^"}\n"^
           "\\hat\\kappa"^(foldr (fn (f,s) => "&"^f^s) "" fields)^
           "\\\\\\hline\n"^
           (foldr (fn ((c1,c2),s) =>
```

```
              "("^c1^","^c2^")"^
              (foldr (fn (f,s') =>
                 "&"^(pp_val (k (c1,c2) f)^s')) "" fields)^
              "\\\\\n"^s)
            "" refs_p)^"\n\\end{tabular}"

        val stream = TextIO.openOut (name^".tex")
        val (z,(si,k)) = solution

    in TextIO.output (stream, (pp_zeta z)^(pp_sigma si)^(pp_kappa k))
       before TextIO.closeOut stream

    end
end (* functor *)
```