Analyzing Java in Isabelle/HOL

— Formalization, Type Safety and Hoare Logic —

David von Oheimb

Institut für Informatik, Lehrstuhl IV

Analyzing Java in Isabelle/HOL

— *Formalization, Type Safety and Hoare Logic* —

David von Oheimb

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Dr. h.c. Jürgen Eickel
Prüfer der Dissertation:		
	1.	Univ.-Prof. Tobias Nipkow, Ph.D.
	2.	Univ.-Prof. Dr. Arnd Poetzsch-Heffter, FernUniversität Hagen

Die Dissertation wurde am 30. November 2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30. Januar 2001 angenommen.

Kurzfassung

Diese Dissertation behandelt die maschinelle Analyse des (fast vollständigen) sequentiellen Teils der objektorientierten Programmiersprache Java. Wir zeigen, dass die Einbettung einer solchen Sprache in einen Theorembeweiser, in diesem Fall Isabelle/HOL, und der Beweis wichtiger metatheoretischer Eigenschaften inzwischen gut möglich ist. Dazu beschreiben wir detailliert die Formalisierung mit Abstrakter Syntax, Typsystem und der Operationellen Semantik sowie ein Anwendungsbeispiel, geben einen Beweis der Typsicherheit, entwickeln eine Axiomatische Semantik und beweisen deren Korrektheit und Vollständigkeit.

Abstract

This thesis deals with machine-checking a large sublanguage of sequential Java, covering nearly all features, in particular the object-oriented ones. It shows that embedding such a language in a theorem prover and deducing practically important properties is meanwhile possible and explains in detail how this can be achieved.

We formalize the abstract syntax, and the static semantics including the type system and well-formedness conditions, as well as an operational (evaluation) semantics of the language. Based on these definitions, we can express soundness of the type system, an important design goal claimed to be reached by the designers of Java, and prove that type safety holds indeed. Moreover, we give an axiomatic semantics of partial correctness for both statements and (side-effecting) expressions. We prove the soundness of this semantics relative to the operational semantics, and even prove completeness. We further give a small but instructive application example.

A direct outcome of this work is the confirmation that the design and specification of Java (or at least the subset considered) is reasonable, yet some omissions in the language specification and possibilities for generalizing the design can be pointed out. The second main contribution is a sound and complete Hoare logic, where the soundness proof for our Hoare logic gives new insights into the role of type safety. To our knowledge, this logic is the first one for an object-oriented language that has been proved complete. By-products of this work are a new general technique for handling side-effecting expressions and their results, the discovery of the strongest possible rule of consequence, and a new rule for flexible handling of mutual recursion.

All definitions and proofs have been done fully formally with the interactive theorem prover Isabelle/HOL, representing one of its major applications. This approach guarantees not only rigorous definitions, but also gives maximal confidence in the results obtained. Thus this thesis demonstrates that machine-checking the design of an important non-trivial programming language and conducting meta-theory on it entirely within a theorem proving system has become a reality.

Not that we are competent in ourselves
to claim anything for ourselves,
but our competence comes from God.

2 Corinthians 3:5

Acknowledgments

I thank my supervisor Tobias Nipkow for giving me the opportunity to work in this interesting project, for his prompt and continuous support, and his well-founded and helpful advice.

I thank Arnd Poetzsch-Heffter for being my referee, and am I indebted to him, Tobias Nipkow, Martin Strecker, Marieke Huisman and the members of the Isabelle theorem proving group in Munich for their comments on draft versions of this thesis. Furthermore I am grateful to Cornelia Pusch, Sophia Drossopoulou, Donald Syme, Martin Hofmann, Peter Müller, Thomas Kleymann and Francis Tang for inspiring and clarifying discussions.

I thank the — former and current — members and guests of our working group, namely Tobias Nipkow, Franz Regensburger, Dieter Nazareth, Christian Prehofer, Konrad Slind, Olaf Müller, Cornelia Pusch, Wolfgang Naraschewski, Markus Wenzel, Leonor Prensa Nieto, John Harrison, Stefan Berghofer, Gerwin Klein, Gertrud Bauer, Sebastian Skalberg, Raman Ramanujam and Giampaolo Bella, as well as Manfred Broy and the remaining colleagues and staff members of our chair, for the excellent working atmosphere and the technical and organizational support.

I thank my mother Erika, Reinhard Kahl, Edith Steiler, my landlords Gottlieb and Ilse Endraß in Gröbenzell, Helmut Kilgus, Marlies Gilliam and the members of the Vineyard church initiative in Puchheim for their encouragement and prayers.

To Hagen and Erika

Contents

Chapter 1

Introduction

1.1 Motivation

Ever since the invention and first serious applications of computers in the middle of the last century, the importance of computer software has been growing enormously. Starting as a by-product of electronic hardware development, meanwhile it has become an industry of its own heavily influencing production, administration, commerce and leisure.

Becoming more and more a crucial part in our modern society, correctness problems and the resulting risks of applying software have emerged gradually but evidently. The most striking example recently has been the well-known "Y2K bug" [NYT99] demonstrating to anyone in the world how easily misconceptions on the use of software and faults in non-tested situations could become a global — at least economic — threat. Other malfunctions have been more spectacular like the destruction of an Ariane V rocket [L$^+$96] or much more annoying to millions of users like the everyday experience of MS Windows bugs [Kar98].

Even long before these and many other failures actually happened, on the October 1968 NATO Science Committee Conference on Software Engineering in Garmisch-Partenkirchen the term "software crisis" was used to describe the situation of software being late, over-budget and full of errors. Unfortunately, this crisis has become chronic [Gib94] since then. The reasons for this phenomenon are evident: the tasks of computation are inherently — and increasingly — complex, and so is the software implementing it. Moreover there is high demand of producing software within very short time, having only a limited number of skilled developers available, and even these are non-perfect human beings.

To remedy the situation, typically more or less rigorous tests are performed, but these are not sufficient as they do not guarantee the absence of errors. A real solution to rule out design and coding — yet not analysis — errors would be to prove correctness using formal (mathematical, logical) methods, for two reasons: formalization helps to understand the problem to be dealt with, and proofs give maximal confidence in correctness. Unfortunately even just applying formal methods is typically quite difficult and takes a lot of effort and thus is rarely used in practice today.

The vision of computer science and hope of at least those parts of industry developing safety- and mission-critical software is that through research and experience formal methods get mature and easier to apply, *e.g.* by automation. Further positive effects can be expected from improved programming languages and related tools. A promising relatively new aspect is to use machine-checked formal methods to analyze such equipment and enhance their development. The work presented here aims to be a step in this direction.

1.2 Aims

1.2.1 Formal Language Semantics

As opposed to informal human languages, languages used for expressing computer programs have a small vocabulary, a simple structure and uniquely definable semantics. The theory of programming languages has a relatively long and successful tradition, as can be seen from the available standard textbooks on the subject, *e.g.* [Win93] and [NN92]. In order to obtain a precise description and analysis, formal techniques originating from discrete mathematics, in particular logics, should be applied (and extended). This is particularly important since even if each of the basic principles of a programming language are normally well understood, their interplay often causes effects that are difficult to comprehend. Without formal analysis, design errors like loopholes in the type system may occur, as happened *e.g.* for Eiffel [Coo89].

1.2.2 Need for Machine Support

Programming language theory typically deals with idealized core languages that are very convenient for studying difficult phenomenons without having to bother with (apparently) unrelated features and details. Up to this point, doing pen-and-paper work is sufficient, even if machine-checked formalizations and proofs increase precision and reliability. Yet when it comes to applying the theory to actual languages, as (partially) done *e.g.* in [DE97a], one is forced to handle a lot of details as well as their sometimes unexpected interference. The sheer amount of work and the number of mistakes that inevitably creep in render the pen-and-paper approach infeasible, in particular in the presence of changes and extensions. Here, machine-supported formalization and proofs come to the rescue, promising correctness and at least partial automation.

1.2.3 Maturity of Machine Support

Practically useful theorem proving systems have been around for more than one decade [Gor85, C$^+$86, OSR92, Pau90, DFH$^+$93, Pol94]. Their standard in terms of correctness and robustness is usually very high (although for PVS [PVS, GH98] efficiency is more important), but at least for several years their expressiveness, degree of automation or scalability has been insufficient for large complex applications. Also the methodology of language embeddings had to be developed, where a major contribution is due to Gordon [Gor89]. Fortunately, over time there has been considerable progress in theorem proving, taking as an example Isabelle [Pau94b, Isa] where support for inductive definitions and mutually recursive datatypes [BW99], combinations of rewriting with proof search strategies, and a convenient user interface [AGKS99] have been added. Progress in the general computing power available plays an important role as well. The overall goal of this thesis is to demonstrate that, making use of all these advances,

machine-checking the design and meta-theory
of realistic programming languages has become feasible.

As a case study supporting this claim we

- take one of the currently most popular programming languages, Java (*cf.* §1.5),
- formalize (*cf.* §1.6) it, and
- conduct meta-theoretical proofs of various properties
- in a state-of-the art verification system, Isabelle/HOL (*cf.* §1.7).

1.2.4 Advances in Methodology

This work is the first larger-scale embedding of an object-oriented programming language for performing meta-theory in a theorem prover. Such a challenging task requires not only a high degree of dedication, but also extensions of the existing methodology of embeddings, in particular applying software-engineering techniques (*cf.* §1.6.4) in order to keep the resulting system manageable. So a secondary aim of this thesis is to convey experience and guidelines supporting future formal investigations of other languages using Isabelle or similar theorem provers. Therefore we present the formal issues not in full mathematical abstraction, but in the more technical style actually required by the theorem prover. On the other hand, we abstract from the peculiarities of Isabelle as far as possible, stressing the general principles that apply also to its relatives.

1.2.5 Design Check

Concerning the embedded language itself, we aim at checking the consistency of Java's design and specification. We do not really expect to find severe loopholes in the language design, in particular concerning type soundness, but one cannot be sure without rigorous verification. Yet we do expect to identify problematic (in particular in the sense of unnecessarily difficult) features and find minor omissions and inconsistencies of the specification. This provides valuable feedback for the designers and specification authors as well as language tool (*e.g.* compiler) developers and programmers who may be warned of potential pitfalls.

1.2.6 Hoare Logic

Next to a proof of type soundness, as the second major meta-theoretical undertaking we aim at developing a Hoare logic (*i.e.* axiomatic semantics) for Java and prove it correct w.r.t. our operational semantics and, if possible, also prove relative completeness. The latter issue is particularly challenging since by now no such completeness proof is known for any object-oriented language, not even on paper.

1.2.7 Direct Applications

Our formalization itself can be — and actually is — used in several ways.

- The formalization of current Java is a good starting point for investigating possible future extensions, for example generic types and intersection types [BW98].
- Together with a formalization of the Java Virtual Machine (JVM) and its bytecode, it serves as the basis for compiler verification.
- The Hoare logic may of course be used to perform program verification.
- Even further in this direction, we plan to formalize program design using UML [BRJ98, EFLR99] and in particular OCL [WK99] for verifying the implementation of high-level specifications.
- A simplified version [NOP00] of the system is used for teaching on semantics.

1.3 Related Work

This section provides an overview of the literature related to our work. Further related work and more detailed remarks on some of the references summarized here will be given in the subsequent chapters where appropriate.

The work described in this thesis is has been performed in the context of the DFG Project Bali [NOPK] aiming to treat the major aspects of Java and its environment formally within Isabelle/HOL. Next to investigations of the Java source language [NO98, ON99] describing previous stages of the work presented here, there is a corresponding formalization of the JVM [Pus98a] and in particular on the Bytecode Verifier [Pus99, KN00]. The papers [OP98, NOP00] give an overview on both the source and bytecode levels. Work on combining them via compiler correctness is in progress.

The first formalization of a (small, later extended) subset of Java is due to Drossopoulou and Eisenbach [DE97a, DE99] including a pen-and-paper proof of type soundness, which heavily inspired our formalization and type soundness proof. Börger and Schulte [BS99] formalize (as an Abstract State Machine) a rather large subset including multi-threading, aiming at compiler verification and rapid prototyping, but no proofs have been reported. Wallace [Wal97] gives a quite similar ASM semantics of Java slightly better structured through the use of so-called Montages.

Concerning the embedding of programming language semantics in a theorem prover, the pioneering work is Gordon's [Gor89], who defined the semantics of a simple `while`-language (as a shallow embedding, *cf.* §1.6.5) and derived Hoare logic rules within his HOL system. Later, Nipkow embedded a similar language called IMP defined for didactic purposes [Win93] in Isabelle/HOL [Nip98]. A recent work on a non-object-oriented language is due to Norrish [Nor98] formalizing the bulk of the C language as a transition semantics in Gordon's HOL, proving type preservation and useful lemmas concerning determinism [Nor99] for it, and deriving a Hoare logic for a subset free of side effects and abrupt termination.

Various sublanguages of Java have been formalized mechanically by

Syme [Sym99b] proving type soundness as the major case study for his DECLARE prover [Sym97, Sym99a], directly mechanizing and improving the work contained in [DE97a],

Jacobs et al. [JBH$^+$98, HJ00] translating Java classes to a more or less denotational representation (as PVS or Isabelle theories) using their LOOP [J$^+$a] tool and performing program verification with a Hoare logic proved sound within their system,

Attali et al. [ACR98, ACR] giving an executable transition semantics and a visualization tool for a large subset of Java within the Centaur system aiming at program simulation and (ultimately) at verification as well.

Recently several proposals for not machine-checked Hoare logics for some more or less object-oriented languages have been given, *e.g.* [AL97, dB99]. The work of Poetzsch-Heffter and Müller [PHM99] dealing with a subset of Java, contains also a soundness proof (on paper) which inspired parts of our soundness proof. Inspiration for our completeness proof and the approach for dealing with auxiliary variable stems from Kleymann[1] [Sch97, Kle98] who gives a logic of total correctness and VDM rules for a simple imperative language with a single recursive procedure and proves soundness and completeness using the theorem prover Lego. Homeier and Martin [HM96] deal with a similar language but including mutual recursion and prove the correctness of a verification condition generator using Gordon's HOL, while von Oheimb [Ohe99] proves also completeness for an IMP-like language with mutual recursion using Isabelle, which was an important building block of this thesis.

[1] formerly Schreiber

1.4 Overview

The structure of this thesis is as follows. The remainder of this introducing Chapter 1 gives background information on Java and some of its derivatives, general formalization issues, and the theorem prover Isabelle/HOL.

Chapters 2 and 3 define the operational semantics of Java$^{\ell ight}$ giving all the details of our formalization. The former focuses on the static aspects, thus syntax, type system, and well-formedness conditions, whereas the latter contains the dynamic aspects, thus the state model and evaluation rules.

Chapter 4 deals with type soundness as the first application of our operational semantics. We explain the necessary notions, comment on our proof, and compare it to an alternative approach using transition semantics.

Chapter 5 introduces our Hoare logic for Java$^{\ell ight}$ including all the techniques required like auxiliary variables, intermediate values, and result entries. We sketch the proofs of soundness and completeness covering two alternatives for handling mutual recursion.

Chapter 6 gives a small instructive example applying most of the concepts introduced in the chapters before.

Chapter 7 concludes by summarizing the achievements of this thesis and the lessons learnt for Java, formalization and theorem proving, and points out further work.

This thesis contains parts of the following previously published material: [NO98, ON99, Ohe99, Ohe00a].

1.5 Java-like Languages

From the early days of programming up to now programmers are accustomed to an imperative (state-oriented, operational) style of programming and using corresponding languages, even though functional and other declarative languages would often be more appropriate and much less error-prone. Nevertheless there has been development over procedural to object-oriented style giving better structuring of software and thus improving reliability, maintainability and reuse. Still at least two problematic issues remain: mutable state, and object references. Introducing side effects and aliasing, they are the main challenges for reasoning about programs. In contrast, they are rather well understood and easy (yet in part cumbersome) to model on the level of language semantics. Other features tend to be difficult on both levels, like subtyping involving inheritance and hiding, exceptions leading to potential unexpected termination, and concurrency involving non-determinism and synchronization.

1.5.1 Java

Java [AGH00] is currently one of the most popular programming languages since it combines recent experience on object-oriented programming with high security and portability. Despite some attempts for public standardization, proprietary modifications and even plagiarism, Java is still under full control of Sun Microsystems. This has the advantage of maximal integrity but on the other hand gives little opportunity for suggestions (*e.g.* from academia) for improvement other than those — directly or indirectly — meeting the economic interests of that company.

None of Java's features had been actually new at the time of its design, but the good mixture by concentrating on useful and more or less well-understood features makes the difference. These are

- (rather) purely object-oriented paradigm, as opposed to *e.g.* C++
- interfaces and classes implementing them
- (almost) static typing and initialization test
- dynamic object allocation and automatic garbage collection
- static overloading and dynamic binding
- elaborate exception handling
- name spaces and visibility control
- concurrency through language-supported multi-threading
- virtual machine [LY96] making compiled code portable

Some of Java's features are problematic and may be criticize as flaws, like

- covariance (instead of non-variance) of array types making dynamic type checks necessary for array assignment
- lack of generic types, where corresponding extensions have been proposed and discussed for several years [AFM97, MBL97, OW97], of which the one by Odersky et al. meanwhile has been selected for realization [BOSW98]
- counterintuitive and hard to implement semantics of concurrency which is currently being revised
- minor unnecessary restrictions, as summarized in §1.5.3

Our work focuses on Java version 1.0, as this was the one available at the beginning of the project. The extensions made meanwhile, in particular inner classes, are not of special interest for us. Thus all our references to the Java specification refer to its first official version [GJS96].

1.5.2 Java Card and Java$^{\ell ight}$

Java is aimed to run not only on conventional personal computers and workstations, but also on embedded systems and so-called *Smart Cards* [Sun99a, TJ97]. Since on smart cards memory (and computational power) is rather limited, a sublanguage of Java that can cope with very little resources, called *Java Card* [Sun99b], has been defined. To this end, garbage collection and multi-threading are not included, and the application programming interface (API) providing platform independence is shortened.

The sublanguage of Java that we consider, called *Java*$^{\ell ight}$, is rather similar to Java Card. There are only two important features still missing, *viz.* name spaces and visibility control, which we aim to include later. Apart from that, we believe to have included all features important for an investigation of the semantics of a practical imperative object-oriented language.

For easier comparison with other approaches formalizing diverse sublanguages of Java, we give a list of the features of Java$^{\ell ight}$:

- class and interface declarations
- class fields and methods
- instance fields and methods

- inheritance, overriding, and hiding
- objects (including arrays)
- a few primitive types
- all relations on reference types
- dynamic object allocation
- static initialization
- static overloading of fields and methods
- dynamic binding of method calls
- exception throwing and handling

Note that these features are more or less orthogonal to each other. In contrast, instances of certain concepts, *e.g.* numerous variants of control statements, can be reduced to a most generic or at least prototypical one. In this case we only provide that generic variant, from which the others can be derived — typically by instantiation — without losing expressiveness.

1.5.3 Differences to Java

This subsection gives a detailed account of how Java$^{\ell ight}$ differs from Java 1.0 and Java Card.

There are several aspects of Java that are not modeled in Java$^{\ell ight}$ (in order to obtain a lean formalization) and cannot be emulated neither:

- multi-threading, a problematic feature also not included in Java Card
- garbage collection, which is also not included in Java Card
- packages and hierarchical name spaces, which we are planning to add later
- field and method modifiers other than `static`, also to be added later
- definite assignment
- `throws` clauses
- (standard) unary and binary operators
- memory overflow when allocating a class object
- stack overflow

Some features of Java that are left out in Java$^{\ell ight}$ for simplicity can be emulated:

- interface fields (*i.e.* named constants), *cf.* §2.6.2
- user-defined constructors, *cf.* §2.6.2
- multiple static initializers, *cf.* §2.6.2
- creation of multi-dimensional arrays, *cf.* §2.4.2
- nested blocks and inner local variables, *cf.* §2.4.1
- static references (by type name) to methods and fields, *cf.* §2.4.2 and 2.4.3
- `break`, `continue`, and `return` statements (to be added later), *cf.* §2.4.1
- `try` _ `catch` _ statement with multiple `catch` clauses, *cf.* §2.4.1
- return statements, *cf.* §2.6.1

As far as Java$^{\ell ight}$ is a subset of Java, it adheres to the Java language specification [GJS96], except for several useful generalizations:

- any expression may be used as a statement, *cf.* §2.4.1
- the `finally` statement may be used in any context, *cf.* §2.4.1
- `super` may be used as an ordinary expression, *cf.* §2.4.2
- in class and interface declarations we allow the result type of a method overriding some method m to widen to the result type of m (instead of requiring it to be identical), *cf.* §2.9.2 and 2.9.3
- in the same way, the relation between methods taken into account when defining the narrowing relation is relaxed, *cf.* §2.7.4
- if a class or an interface inherits more than one method with the same signature, these methods need not have identical return types as long as they are overridden or implemented by a method whose return type is a common subtype, *cf.* §2.9.2 and 2.9.3
- we use a weaker notion of "more specific methods", which seems more appropriate from the software engineering viewpoint and extends the set of legal method invocations, *cf.* §2.8.5
- the type of an assignment is determined by the right-hand side, which can be more specific than the left-hand side, *cf.* §2.8.4

We found several issues not specified in [GJS96] and defined a reasonable behavior that seems to be consistent with current implementations:

- given a Null reference, the `throw` statement raises a `NullPointer` exception, *cf.* §3.2.5
- each standard exception thrown yields a fresh exception object, *cf.* §3.1.3
- if there is not enough memory even to allocate an `OutOfMemory` error, program execution simply halts, *cf.* §3.2.8
- the exact position of class initializations has been fixed, *cf.* §3.2.6

A few parts of the specification are even misleading and had to be corrected:

- methods of class `Object` may be called upon any interface, *cf.* §2.8.5
- method invocation mode `super` is effectively static, *cf.* §3.2.9
- method lookup does not need to take the return type into account, *cf.* §3.2.9

1.6 Formalization

Even rather abstract entities like programming languages (except for their syntax) are typically defined in an informal — but hopefully precise — way. To make then amenable to formal analysis means to describe them by mathematical means, at least on paper, or — even more rigorously and reliably — on the computer. As formalization is the most creative and interesting part of work like ours, it will take up most of the space in this thesis.

1.6.1 Validation Problem

Unfortunately the crucial transformation step of formalization is itself informal, so much care is required in order to obtain something really useful. Basic validation is done by manual inspection, which in the ideal case is done by several people independently, possibly aided by a tool re-translating the model to a less formal (and more verbose) description. Further steps are identification and proof of key properties of the model, as well as generation of test examples covering as much of the model as possible and comparing their outcome to the original system, for which rapid prototyping and code generation tools are very helpful.

1.6.2 Goals

For a formalization like ours it is important to aim at the following general design goals.

understandability (including readability on the lexical level) is the basic requirement, as otherwise handling and applying the formalized model will be severely hampered

faithfulness to the original specification as otherwise the properties derived from the model do not apply to the original system

maintainability for ease of change and extension

executability for validation checks, rapid prototyping, and didactic purposes

adequacy for applications, in particular for theorem proving and code generation

It is worthwhile to keep these goals in mind while reading the subsequent chapters and to judge yourself how far we have reached them.

1.6.3 Principles

In order to achieve the goals mentioned above, principles successfully used in ordinary software development apply also to language embeddings:

simplicity and succinctness: keeping things as short and simple as possible minimizes errors and eases validation as well as applications

standardization: using symbols, terms and concepts that are generally well-known and keeping names consistent with the original specification of the system being modeled avoids confusion and improves maintainability and suitability for code generation.

modularity and locality: grouping together things that belong to each other intrinsically enhances maintainability

abstraction and minimal redundancy: avoiding unnecessary details and replications of any kind (unless put to the extreme) strongly improves maintainability and adequacy for theorem proving

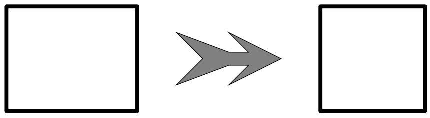
All of these principles promote the basic goal of understandability as well.

1.6.4 Techniques

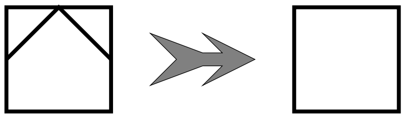
There are several distinct techniques aiming at the goals (by implementing the principles) given. Each situation within our formalization where we apply them is marked with a tag referring to the following list.

The first group of techniques concerns shaping the outer boundaries of the model, *i.e.* describes a certain relation to the original language specification:

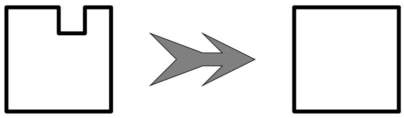
Restrictions by leaving out features that are too difficult or uninteresting



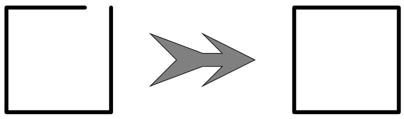
Underspecification by leaving out irrelevant details



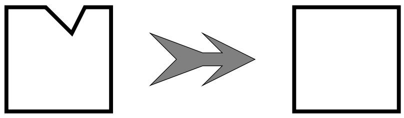
Generalization by removing unnecessary restrictions



Clarification by filling gaps in the specification

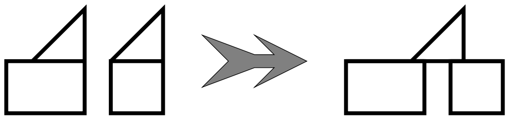


Correction of misleading parts of the specification

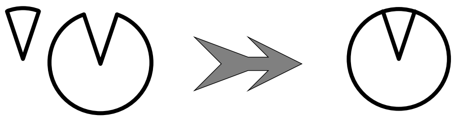


The second group of features concern the inner structure of the model:

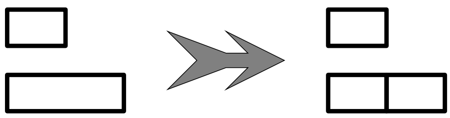
Factoring[2] by forming hierarchies capturing common behavior



Unification of related concepts



Reduction to already defined concepts



[2]A semi-graphical pun by Wenzel: "Factory$_1$ + Factory$_2$ $\rightsquigarrow$ Factored Factories ;-)"

1.6.5 Embedding Style

When formalizing a programming language in a theorem prover, which is also called *embedding*, one has basically two options [BGG+92]:

deep embeddings define as a first step the (abstract) syntax of the language and then assign semantics to it. This is useful when doing meta-theory in the language since one can express properties of the syntactic structure and prove generic properties of the language (such as type soundness).

shallow embeddings define the semantics of each construct in the language by an entity within the underlying logic. This is advantageous for reasoning about individual programs of the language as the extra syntactic level is avoided.

As our aim is meta-theory, from this characterization it should be clear that we mainly use a deep embedding. It is also possible to mix both styles, which we will do when shallowly embedding the assertion language of our axiomatic semantics into higher-order logic.

1.7 Isabelle/HOL

For our formalization and proofs we use the theorem proving system *Isabelle/HOL*. This is the generic interactive theorem prover Isabelle [Pau94b] instantiated with Church's classical, simply-typed Higher-Order Logic (HOL) [Chu40, And86]. Isabelle/HOL has been developed by Paulson in Cambridge and Nipkow in Munich. The system is evolving further, in particular through the work of the Munich theorem proving group (Wenzel et al.) [Isa].

The merits of Isabelle/HOL we heavily make use of are:

expressiveness of the logic enables natural and direct formalization

type inference helps to easily detect ill-formed expressions

syntax capabilities allow well-readable formulas

proof tools suitable for rather convenient semi-automatic reasoning

trustworthiness of the underlying inference machinery due to the "LCF approach" [GMW79, Pau87]

convenient use achieved with the user interface Proof General [Asp00a, Asp00b]

user support through extensive documentation and the developers themselves

Recently Isabelle has acquired a new metalanguage for declarations and proofs (replacing ML [MTH90] as the proof script language) called Isar [Wen99b, Wen99a], yet we always refer to the "traditional" version of the Isabelle user interface, as introduced *e.g.* in [Pau94a]. A more recent gentle introduction to Isabelle/HOL is [Nip99]. Here and in the subsequent chapters by *HOL* we mean Isabelle/HOL, which is not to be confused with its nearest relative, Gordon's HOL system [GM93], currently referred to as HOL98 [HOLb].

1.7.1 Presentation of Formal Content

In the remainder of this section, we will briefly introduce those aspects of Isabelle required for understanding the definitions and theorems given in the subsequent chapters. According to our aim to give a technically oriented, but not prover-specific presentation, we abstract from syntactic details by leaving out the respective Isabelle keywords, quotes, *etc.* The actual Isabelle theory sources are given in the appendix, and the full sources including the proof scripts are available on the web at `http://isabelle.in.tum.de/Bali/src/Bali5/`.

We heavily make use of Isabelle's capability to use common "mathematical" syntax for many logical connectives and other symbols, *e.g.* '∀' instead of ASCII replacements like 'All' or '!'. Not only are graphical symbols much more readable, but they also form the tokens of a generally understood mathematical notation, as opposed to any prover-specific pure-ASCII transcription.

We further adopt the following typographic conventions: The names of logical constants like 'True' or 'cfield' appear in sans serif, while type names like '*bool*' or '*state*' and variables like '*v*' appear in italics, and Java keywords like '`catch`' appear in typewriter font.

1.7.2 Theories

Isabelle source is structured as a directed acyclic graph of *theories*, each comprising a list of sections. A section may introduce a piece of syntax, a type or a logical constant, or define their properties.

Types follow the syntax of ML. Type abbreviations are introduced simply as equations. Logical *constants* are declared by giving their name and type, separated by '::'. Both types and constants may obtain additional general mixfix syntax (possibly with graphical symbols and user-defined precedences). We write non-recursive definitions as well as syntactic abbreviations (which should enhance readability of long formulas) with '≡'.

The appearance of formulas is fairly standard, *e.g.* '⟶' is the (right-associative) infix implication symbol. Terms are expressions of an extended λ-calculus similar to ML. Function application is written in curried style. Predicates are functions with Boolean result.

Primitive recursive function definitions are written with pattern matching as usual. There are also general (but well-founded, since HOL is a total logic) recursive definitions. We will heavily use inductive relations and datatypes. A free *datatype* is defined by listing its constructors together with their argument types, separated by '|'. All these higher forms of definitions are handled by separate packages that produce more low-level syntax, type and constant definitions and derive their characteristic properties. For instance, the datatype declaration

$$\begin{aligned} t = {} & C\ t_1\ \ t_2 \\ & \mid \ldots \end{aligned}$$

induces a declaration for the new type t and the injective function $C :: t_1 \to t_2 \to t$ as datatype *constructor*. Additionally, for our presentation we assume[3] that for each constructor $C :: t_1 \to t_2 \to t$ there is a corresponding *destructor* (a.k.a. selector) $\mathsf{the\_}C :: t \to t_1 \times t_2$, defined as $\mathsf{the\_}C\ z \equiv \varepsilon(x,y).\ z = C\ x\ y$. For descriptions like these we apply Hilbert's choice operator ε, where $\varepsilon x.\ P\ x$ denotes some (deterministically chosen) value x satisfying P, or some default value if no such value exists. The default value is represented as the polymorphic constant $\mathsf{arbitrary} :: \alpha$ and is completely unspecified and thus definable as $\varepsilon x.\ \mathsf{False}$. Note that in HOL there is no empty type.

[3]The current HOL datatype package does not provide destructors, but we define them manually where needed.

1.7.3 HOL Library

We build our formalization on top of the standard HOL library. Here we briefly introduce the types and functions used later. The fundamentals of HOL are introduced in [NPW94], whereas the latest version of HOL can be found in [HOLa].

In HOL there are the basic types *unit* (with a single value denoted by ()), *bool*, *int* and *nat*, as well as the polymorphic type $(\alpha)set$ of homogeneous sets for any element type α. Occasionally we apply the infix 'image' operator lifting a function over a set, defined as $f\text{``}\, S \equiv \{f\ x\ |x.\ x{\in}S\}$.

The product type $\alpha \times \beta$ comes with the projection functions fst and snd. Tuples are (improper) pairs nested to the right, *e.g.* $(a,b,c) = (a,(b,c))$. They may be used also as patterns like in $\lambda(x,y).\ f\ x\ y$. Sometimes we use tuples where (extensible) records with named fields would be more appropriate, which is for historical reasons since records were not available in HOL at earlier stages of our formalization work. Meanwhile there is even a methodology for object-oriented verification using extensible records [NW98], which is aimed to support program verification using shallow embedding.

The sum type $\alpha + \beta$ comes with the injections Inl and Inr. For the ternary sum $\alpha + \beta + \gamma$ we assume the injections $\mathsf{In1}$, $\mathsf{In2}$ and $\mathsf{In3}$.. Furthermore, for the nested sums that we will use the abbreviations $\mathsf{In1l}\ e \equiv \mathsf{In1}\ (\mathsf{Inl}\ e)$ and $\mathsf{In1r}\ e \equiv \mathsf{In1}\ (\mathsf{Inr}\ e)$ are handy.

The list type $(\alpha)list$ is defined via its constructors [] denoting the empty list and the infix 'cons' operator '#', which are introduced by the datatype declaration

$$(\alpha)list = []\ |\ \alpha\#(\alpha)list$$

The concatenation operator on lists is written as the infix symbol '@'. We use the functional $\mathsf{map} :: (\alpha \to \beta) \to (\alpha)list \to (\beta)list$ applying a function to all elements of a list, the function $\mathsf{length} :: (\alpha)list \to nat$, a check for absence of duplicate elements $\mathsf{nodups} :: (\alpha)list \to bool$, and a conversion function $\mathsf{set} :: (\alpha)list \to (\alpha)set$.

We frequently use the datatype

$$(\alpha)option = \mathsf{None}\ |\ \mathsf{Some}\ \alpha$$

It has an unpacking function $\mathsf{the} :: (\alpha)option \to \alpha$ such that $\mathsf{the}\ (\mathsf{Some}\ x) = x$ and $\mathsf{the}\ \mathsf{None} = \mathsf{arbitrary}$. There is a simple function mapping $\mathsf{o2s} :: (\alpha)option \to (\alpha)set$ converting an optional value to a set, with the characteristic equations $\mathsf{o2s}\ (\mathsf{Some}\ x) = \{x\}$ and $\mathsf{o2s}\ \mathsf{None} = \emptyset$. With its help, we define the bounded quantifications $\forall x{\in}A\colon P \equiv \forall x{\in}\mathsf{o2s}\ A.\ P$ and $\exists x{\in}A\colon P \equiv \exists x{\in}\mathsf{o2s}\ A.\ P$. Analogous to the map function for lists, there is

$$\begin{array}{l} \mathsf{option\_map} :: (\alpha \to \beta) \to ((\alpha)option \to (\beta)option) \\ \mathsf{option\_map}\ f \equiv \lambda z.\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{None} \to \mathsf{None}\ |\ \mathsf{Some}\ x \to \mathsf{Some}\ (f\ x) \end{array}$$

1.7.4 Proof Tools

Proofs with (traditional) Isabelle are conducted in a goal-directed, backward-chaining style: goals are stated and successively reduced to further subgoals using so-called *tactics*, which are transformers on the *proof state*, until the proof state becomes trivial. Tactics may be combined using *tacticals*, which act as control structures.

The basic tactics are *resolution*, applying inference rules backwards in natural deduction style, and *rewriting*, applying (conditional) directed equalities. Natural deduction involving search with backtracking can be automated using the so-called *classical reasoner*, and rewriting typically is done with the *simplifier*, both using suitable sets of rules. These two tools are often combined.

A typical proof is semi-automatic in the sense that the user directs the important higher-level steps like inductions, case distinctions, and classical proof schemes. On the less interesting and more straightforward remaining subgoals of the proof, the automatic tools are often smart enough to find suitable proofs, at least when equipped with lemmas expressing characteristic properties of the entities involved. Such lemmas typically include obvious introduction and elimination or rewrite rules, which an experienced user can predict and prove in advance or at least identify on demand.

Chapter 2

Static Semantics

As the basis for all further investigations, we formalize the static aspects of Java$^{\ell ight}$, *i.e.* its (abstract) syntax, type system, and well-formedness conditions. Many of our definitions have been inspired by Drossopoulou and Eisenbach's pioneering work [DE97a], yet improved and adapted for our needs.

As is typical for formal systems, we have to define all features in bottom-up fashion and our presentation aims to reflect this faithfully. Doing so we follow a strictly layered approach, starting with simple syntactic properties for all features of the language and stepwise moving up to the most complex context-related semantic properties. One might wonder why we do not pursue a more feature-oriented approach, *i.e.* getting through the different layers separately for each construct. This is partially done in the Java Specification [GJS96], but impossible in our rigorous setting not allowing forward references: for example, even for a rather simple language feature like the fields of a class, the main well-formedness condition is that the field types have to exist, which can be checked only in the context of the whole program, which cannot be defined without all other features already being known at least syntactically. A similar obstacle is *e.g.* the subclass relation, which has to be defined in the context of whole programs but is already used in well-formedness conditions of methods. Thus one has to take care to minimize forward references in the model and avoid circularities.

An actual pitfall have been the definitions of lookup functions for methods and fields together with the definition of well-formed classes as given in [DE97a, §2.4]. They are implicitly circular [Sym99b, §2.1] because the lookup functions are well-defined only if the subclass hierarchy is well-founded (*cf.* §2.6.4), but on the other hand the definition of well-formedness makes use of the lookup functions.

2.1 Names

Java distinguishes between the primitive notion of *identifiers* [GJS96, §3.8] and the derived notion of *names* [GJS96, §6.2, 6.5]. Since in Java$^{\ell ight}$ we do not consider packages [Restriction □ ⋙ □], we may take names and identifiers as synonyms.

We model names in HOL (basically) as opaque (*i.e.* not further defined) types [Underspec ◸ ⋙ □] as our name space is flat and thus does not have any interesting internal structure. Thus we introduce the type *mname* of *method names* without further specifying it.

The second kind of names is *expression names*, *i.e.* names of fields and local variables, modeled by the opaque type *ename*.

The third kind is *type names*. Since there are some language-defined type names like `Object`, we define a free datatype *tname* with special constructors for the distinct names and a default constructor for the remaining type names, denoted by *tnam*:

$$
\begin{aligned}
\textit{tname} &= \texttt{Object} \\
&\mid \ \mathsf{SXcpt}\ \textit{xname} \\
&\mid \ \mathsf{TName}\ \textit{tnam} \\
\textit{xname} &= \texttt{Throwable} \\
&\mid \ \texttt{NullPointer} \mid \texttt{OutOfMemory} \mid \texttt{ClassCast} \\
&\mid \ \texttt{NegArrSize} \ \mid \texttt{IndOutBound} \mid \texttt{ArrStore}
\end{aligned}
$$

The constructor SXcpt is used to comprise the names of the language-defined exceptions [Factoring ≫].

2.2 Types

In Java^{light}, we have all reference types (and the null type) of Java, but only the most important primitive types [Restriction ≫].

$$
\begin{aligned}
\textit{ty} \quad &= \mathsf{PrimT}\ \textit{prim\_ty} \\
&\mid \mathsf{RefT}\ \ \textit{ref\_ty}
\end{aligned}
$$

Each of both type hierarchies is conveniently described using a HOL datatype.

2.2.1 Primitive Types

In our model there are three kinds of *primitive types* [GJS96, §4.2]: Boolean values, integers, and the void type. The latter is an artifact useful as dummy return type of void methods.

$$
\begin{aligned}
\textit{prim\_ty} &= \texttt{void} \\
&\mid \texttt{boolean} \\
&\mid \texttt{int}
\end{aligned}
$$

2.2.2 Reference Types

In Java there are three kinds of *reference types* [GJS96, §4.3]: interface, class, and array types. Additionally, we consider the *null type* [GJS96, §4.1] as being a (proper) reference type [Unification ≫].

The HOL datatype *ty* stands for any Java type. Note that it has to be defined by mutual recursion with *ref_ty* since both datatypes depend on each other.

$$
\begin{aligned}
\textit{ref\_ty} &= \mathsf{NullT} \\
&\mid \mathsf{IfaceT}\ \ \textit{tname} \\
&\mid \mathsf{ClassT}\ \ \textit{tname} \\
&\mid \mathsf{ArrayT}\ \ \textit{ty}
\end{aligned}
$$

Though it is helpful to group together the reference types using the constructor RefT [Factoring ≫], it is often convenient to refer to them directly with simple names. Therefore we add the following abbreviations.

$$
\begin{aligned}
\mathsf{NT} \quad &\equiv \mathsf{RefT}\ \ \ \mathsf{NullT} \\
\mathsf{Iface}\ I \ &\equiv \mathsf{RefT}\ (\mathsf{IfaceT}\ I) \\
\mathsf{Class}\ C &\equiv \mathsf{RefT}\ (\mathsf{ClassT}\ C) \\
T[] \quad &\equiv \mathsf{RefT}\ (\mathsf{ArrayT}\ T)
\end{aligned}
$$

2.3 Values

We have to define the notion of a *value* already here (within the static rather than dynamic semantics) since our model of literal terms relies on it.

The definition of values relies in a straightforward way on the standard HOL types of Boolean values *bool* and integers *int*.

$$
\begin{aligned}
val\ =\ & \mathsf{Unit} \\
\mid\ & \mathsf{Bool}\ bool \\
\mid\ & \mathsf{Intg}\ \ int \\
\mid\ & \mathsf{Null} \\
\mid\ & \mathsf{Addr}\ loc
\end{aligned}
$$

The value Unit serves as a dummy result of statements and `void` methods [Unification $\gg$]. The type *loc* of *locations* or *addresses*, *i.e.* non-null references to objects, is not further specified [Underspec $\gg$]. Note that the type *int* is actually too abstract because it allows arbitrarily large integer values, which is of course not the case within Java.

Values normally have an associated type, which we compute with the function typeof by case distinction, a degenerate form of primitive recursion. Since addresses have a type only if they point to an existing object, we make the outcome of typeof optional and in this case depend on a function of HOL type

$$\mathsf{dyn\_ty} = loc \rightarrow (ty)option$$

to be supplied as an extra argument.

$$
\begin{aligned}
&\mathsf{typeof} :: \mathsf{dyn\_ty} \rightarrow val \rightarrow (ty)\ option \\
&\mathsf{typeof}\ dt\ \ \mathsf{Unit} \qquad = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{void}) \\
&\mathsf{typeof}\ dt\ (\mathsf{Bool}\ b) = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{boolean}) \\
&\mathsf{typeof}\ dt\ (\mathsf{Intg}\ i)\ \ = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{int}) \\
&\mathsf{typeof}\ dt\ \ \mathsf{Null} \qquad = \mathsf{Some}\ \mathsf{NT} \\
&\mathsf{typeof}\ dt\ (\mathsf{Addr}\ a) = dt\ a
\end{aligned}
$$

Vice versa, for all types a default value is defined, again by case distinction[1].

$$
\begin{aligned}
&\mathsf{default\_val} :: ty \rightarrow val \\
&\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{void}) \qquad = \mathsf{Unit} \\
&\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{boolean}) = \mathsf{Bool\ False} \\
&\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{int}) \qquad\ = \mathsf{Intg}\ 0 \\
&\mathsf{default\_val}\ (\mathsf{RefT}\ \ \ r) \qquad\quad = \mathsf{Null}
\end{aligned}
$$

2.4 Terms

We distinguish three kinds of Java *terms*: statements, expressions and variables. We define their abstract syntax as the constructors of three (mutually recursive) HOL datatypes.

2.4.1 Statements

Statements [GJS96, §14] in Java$^{\ell ight}$ are reduced to their bare essentials.

For control transfer, we provide only `if` and `while` in their most general forms without the obvious syntactic variants. We do not formalize the `switch` statement and any kind of jumps, *i.e.* `break` and `continue`, labels, and `return` statements [Restriction $\gg$].

[1]Actually, for technical reasons default_val is defined together with an auxiliary function *defpval* for the default value of primitive types.

These have to be emulated with the given conditional and loop statements [2]. Yet if one aims to support them more conveniently for the user, it should be straightforward to handle them using the exception mechanism [Reduction ≫], which would be an optimization of the approach given in [HJ00]: the type of exceptions (*cf.* §3.1.3) just has to be generalized to include the three additional sources of *abrupt completion* [GJS96, §14.1], and suitable catching constructs have to be added to the evaluation rules for loops (*cf.* §3.2.4) and method calls (*cf.* §3.2.9).

We do not consider nested blocks [Restriction ≫]. Thus all local variables have to be declared at the outermost (*i.e.* method) level, and a *block* [GJS96, §14.2] is nothing but a statement that may contain other statements via sequential composition.

stmt = Skip
| Expr *expr*
| *stmt*; *stmt*
| `if`(*expr*) *stmt* `else` *stmt*
| `while`(*expr*) *stmt*
| `throw` *expr*
| `try` *stmt* `catch` (*tname ename*) *stmt*
| *stmt* `finally` *stmt*
| init *tname*

We denote the *empty statement* by Skip and model *expression statements* using Expr which converts expressions to statements causing evaluation for side effects only. Assignments and method calls, which are expressions because they yield a value, can be turned into statements this way. In contrast to Java, for simplicity we allow this for any kind of expression [Generalization ≫].

In order to simplify the semantical rules, we divide the `try _ catch _ finally _` statement into a `try _ catch _` statement and a `_ finally _` statement, which now might be used in any context [Generalization ≫] and is equivalent to a `try _ catch _ finally _` statement with empty `catch` clause. Our version of the `try _ catch _` statement has exactly one `catch` clause [Restriction ≫]. (Multiple `catch` clauses can be emulated with nested `try _ catch _` statements or conditional statements applying the `_ instanceof _` operator.) We model *class initialization* [GJS96, §12.4] with the artificial statement init [Unification ≫] that is inserted at all points where some class is potentially initialized. See §3.2.6 for further description.

Since we do not consider multi-threading [Restriction ≫], the `synchronized` statement [GJS96, §14.17] is not included here.

2.4.2 Expressions

Java *expressions* [GJS96, §15], in particular method calls and object creation, do not only yield values but typically also cause side effects on the program state. A common modeling technique is to get rid of side effects by transforming the problematic expressions into a series of assignments (which are then considered as statements) to temporary variables. We believe that such a transformation is inadequate since it severely alters the structure of programs and has non-trivial semantical connections: consider for example the case of a complex Boolean expression in a loop condition. Not only a number of temporary variables have to be declared and used, but also all intermediate assignment statements that are then necessary have to be inserted twice: before the loop and at the end of the loop body. Care has to be taken to patch `continue` statements accordingly. Further potential pitfalls are

[2]Techniques for doing so are known from computability theory

possible changes to the evaluation order and the exact flow of exception propagation. To avoid all this, we handle expressions first-class, even if this causes inconveniences, above all for the axiomatic semantics (*cf.* §5.1.3).

Our Java$^{\ell ight}$ model leaves out the standard unary and binary operators as their typing and semantics is straightforward [Restriction ≫]. Increment-like operators could be emulated with assignments. We restrict array creation to a single dimension [Restriction ≫]. Creation of multi-dimensional arrays can be emulated with nested array creation, while access and assignment to multi-dimensional arrays is nested anyway.

$$
\begin{array}{rl}
\mathit{expr} = & \texttt{new}\ \mathit{tname} \\
| & \texttt{new}\ \mathit{ty}[\mathit{expr}] \\
| & \mathsf{Cast}\ \mathit{ty}\ \mathit{expr} \\
| & \mathit{expr}\ \texttt{instanceof}\ \mathit{ref\_ty} \\
| & \mathsf{Lit}\ \mathit{val} \\
| & \texttt{super} \\
| & \mathsf{Acc}\ \mathit{var} \\
| & \mathit{var}\ \texttt{:=}\ \mathit{expr} \\
| & \mathit{expr}\ \texttt{?}\ \mathit{expr}\ \texttt{:}\ \mathit{expr} \\
| & \{\mathit{ref\_ty},\ \mathit{ref\_ty},\ \mathit{inv\_mode}\}\mathit{expr}..\mathit{mname}(\{(\mathit{ty})\mathit{list}\}(\mathit{expr})\mathit{list}) \\
| & \mathsf{Methd}\ \mathit{tname}\ \mathit{sig} \\
| & \mathsf{Body}\ \mathit{tname}\ \mathit{stmt}\ \mathit{expr}
\end{array}
$$

Literal values are encoded directly by their semantics [Reduction ≫] with the constructor Lit receiving an argument of type *val* (*cf.* §2.3). *Variable access* uses an explicit constructor, Acc, to avoid syntactic ambiguities. The `this` expression is modeled as a special non-assignable local variable named This [Reduction ≫]. We model `super` as an expression of its own (with the same value as This and the supertype of This as its type) [Generalization ≫]. Though the conditional expression `_ ? _ : _` is quite redundant with the conditional statement, we include it to demonstrate that our model does not suffer from the Conditional Problem (in connection with transition semantics) described in §4.7.3.

Method calls could be represented basically as terms of the form $\mathit{expr}..\mathit{mname}((\mathit{expr})\mathit{list})$. Yet we add, enclosed in braces $\{\ldots\}$, several subterms called *type annotations*. They are not part of the input language but serve as auxiliary information (computed by the type checker) that is crucial for resolving method overloading and for the static binding of fields. Distinguishing between the actual input language and the augmented language, as done by Drossopoulou and Eisenbach [DE99] would lead to a considerable amount of redundancy in the syntax and in particular in the typing rules. We avoid this [Unification ≫] by assuming that the annotations are added beforehand by some preprocessor. Then the correctness of the annotations can be thought of being checked by the typing rules, *cf.* §2.8.5.

The type *inv_mode* gives the *invocation mode* [GJS96, §15.11.3] for a method call and is defined as

$$
\begin{array}{rl}
\mathit{inv\_mode} = & \mathsf{Static} \\
| & \mathsf{Super} \\
| & \mathsf{IntVir}
\end{array}
$$

where IntVir stands for `interface` or `virtual`. Using *inv_mode*, we handle all variants of method calls with a single rule [Unification ≫]. This avoids much redundancy as contained in other approaches with the drawback of complicating the evaluation rule for method call, which now has to cater for all cases. (Of course, simpler rules for specific cases are easily derivable.) The special case of calling a static method from a type name may be emulated by providing (instead of the type name) a null pointer that is cast to the desired type [Reduction ≫]. We define the abbreviation StatRef to assist with this:

$\mathsf{StatRef}\ \mathit{rt} \equiv \mathsf{Cast}\ (\mathsf{RefT}\ \mathit{rt})\ (\mathsf{Lit\ Null})$

The type *sig* stands for the *signature* [GJS96, §8.4.2] of a method identifying it uniquely within a given class. It consists of the method name and the list of its parameter types, not including the result type:

$$sig = mname \times (ty)list$$

The auxiliary expression $\mathsf{Methd}\ C\ m$ is employed within method calls (*cf.* §3.2.9). It denotes the implementation of method m of class C, a concept crucial for the axiomatic semantics, as will be motivated in §5.6.8. The unfolded version of a method implementation is its actual body, for which we introduce $\mathsf{Body}\ D\ c\ e$ where D is the defining class, c is the (block of) statements in the body, and e is the result expression, as motivated in §2.6.1. This further auxiliary term is a useful abstraction used for simplifying the *Methd* rule of the axiomatic semantics.

2.4.3 Variables

Variables can be thought of as mutable expressions. Typically (and this was also our initial solution) they are modeled with expressions for reading access and statements or expressions for assignment. But since in Java there are several (and rather complex) kinds of variables and distinguishing between access and assignment leads to syntactical and semantical redundancy, we model them as a concept of its own [Factoring $\twoheadrightarrow$], which is also more consistent with the Java specification [GJS96, §15.2, 15.25]. This decision complicates the semantical view of variables (*cf.* §3.2.10), yet to a tolerable extent.

There are actually five kinds of variables: local variables, method parameters, instance variables, class variables (*i.e.* static fields), and array components. Local variables and method parameters are unified easily, and even the two kinds of field variables may be combined [Unification $\twoheadrightarrow$], introducing a Boolean tag indicating static fields. One could even consider reducing local variables to field variables, as discussed in §3.1.1.

$$\begin{aligned} var = {} & \mathsf{LVar}\ lname \\ & \mid \{tname, bool\}\, expr..ename \\ & \mid expr[expr] \end{aligned}$$

The terms in braces for field variables are again type annotations, whose exact purpose will be explained in §2.8.6. Analogously to static method calls, the special case of accessing a static field from a type name may be emulated using the abbreviation $\mathsf{StatRef}$ [Reduction $\twoheadrightarrow$].

Local variables (including parameters) have the explicit constructor LVar, which (analogously to Acc) avoids syntactic ambiguities. Its argument is either the name of a proper local variable (with constructor EName) or This where the property $\forall n.\ \mathsf{This} \neq \mathsf{EName}\ n$ is crucial.

$$\begin{aligned} lname & = ename + unit \\ \mathsf{EName} & \equiv \mathsf{Inl} \\ \mathsf{This} & \equiv \mathsf{Inr}\ () \end{aligned}$$

We provide the abbreviations

$$\begin{aligned} \mathtt{this} & \equiv \mathsf{Acc}\ (\mathsf{LVar}\ \mathsf{This}) \\ !!v & \equiv \mathsf{Acc}\ (\mathsf{LVar}\ (\mathsf{EName}\ v)) \\ v\mathtt{:==}e & \equiv \mathsf{Expr}\ (\mathsf{LVar}\ (\mathsf{EName}\ v)\ \mathtt{:=}\ e) \end{aligned}$$

for convenience. The special syntax for a field *f* of the current class, *viz.* leaving out `this`, is not supported [Reduction $\twoheadrightarrow$].

2.4.4 Combination

For most semantical judgments and their properties there are rather similar variants for statements, expressions, expression lists, and variables. In order to allow more uniform propositions which avoids much redundancy, we group them [Factoring] according to their kinds of results using the nested sum type

$term = (expr{+}stmt)\ +\ var\ +\ (expr)list$

Here expressions and statements are combined [Unification] rather tight since when viewing statements as expressions yielding a dummy result, both kinds of terms yield exactly one result value.

We define the auxiliary predicate

$\mathsf{is\_stmt}\ t \equiv \exists c.\ t = \mathsf{In1r}\ c$

2.5 Lookup Tables

A crucial ingredient of our formalization is the type of *lookup tables*, which we model as (pseudo-)partial[3] functions.

2.5.1 Unique Tables

A table[4] with key type α and entry type β can be defined as

$(\alpha,\beta)table\ =\ \alpha \to (\beta)option$

Thus $t\ x = \mathsf{None}$ means that there is no entry for key x, and $t\ x = \mathsf{Some}\ y$ means that x is associated with the entry y.

The empty table, pointwise update, update by lists (of identical length in our applications), combination of tables, and extension of one table by another, are defined as follows:

$$
\begin{array}{ll}
\mathsf{empty} & :: (\alpha,\beta)table \\
\_(\_{\mapsto}\_) & :: (\alpha,\beta)table \to \alpha \to \beta \to (\alpha,\beta)table \\
\_(\_[{\mapsto}]\_) & :: (\alpha,\beta)table \to (\alpha)list \to (\beta)list \to (\alpha,\beta)table \\
\_\ (+)\ \_ & :: (\alpha,\gamma)table \to (\beta,\gamma)table \to (\alpha{+}\beta,\gamma)table \\
\_{+}{+}\_ & :: (\alpha,\beta)table \to (\alpha,\beta)table \to (\alpha,\beta)table
\end{array}
$$

$$
\begin{array}{l}
\mathsf{empty}\ \equiv \lambda k.\ \mathsf{None} \\
t(a{\mapsto}b)\ \equiv \lambda k.\ \mathsf{if}\ k = a\ \mathsf{then\ Some}\ b\ \mathsf{else}\ t\ k \\
t([]\qquad[{\mapsto}][]\qquad) = t \\
t(x\#xs[{\mapsto}]y\#ys) = t(x{\mapsto}y)(xs[{\mapsto}]ys) \\
s\ (+)\ t\ \equiv \lambda k.\ \mathsf{case}\ k\ \mathsf{of\ Inl}\ x \to s\ x \mid \mathsf{Inr}\ y \to t\ y \\
s\ {+}{+}\ t\ \equiv \lambda k.\ \mathsf{case}\ t\ k\ \mathsf{of\ None} \to s\ k \mid \mathsf{Some}\ x \to \mathsf{Some}\ x
\end{array}
$$

We will further need an auxiliary predicate relating two tables by stating that whenever an entry in one table has the same key as an entry in the other table (which models hiding or overriding) the given relation holds:

$$
\begin{array}{l}
\_\ \mathsf{hiding}\ \_\ \mathsf{entails}\ \_\ :: (\alpha,\beta)table \to (\alpha,\gamma)table \to (\beta \to \gamma \to bool) \to bool \\
t\ \ \mathsf{hiding}\ s\ \mathsf{entails}\ R \equiv \forall k.\ \forall x{\in}t\ k{:}\ \forall y{\in}s\ k{:}\ R\ x\ y
\end{array}
$$

[3] Note that from the HOL perspective, all functions are total.

[4] A variant called map meanwhile has been included in the Isabelle/HOL library.

2.5.2 Non-functional Tables

For forming the union of a set of tables, we also need the type of tables that may return multiple entries for a given key,

$$(\alpha,\beta)tables \;=\; \alpha \to (\beta)set$$

together with a union operator and straightforward variants of two of the notions defined above:

$$\begin{array}{ll} \mathsf{Un\_tables} & :: ((\alpha,\beta)tables)set \to (\alpha,\beta)tables \\ \mathsf{Un\_tables}\ ts & \equiv \lambda k.\ \bigcup t{\in}ts.\ t\ k \end{array}$$

$$\begin{array}{ll} \_\oplus\oplus\_ & :: (\alpha,\beta)tables \to (\alpha,\beta)tables \to (\alpha,\beta)tables \\ s \oplus\oplus t & \equiv \lambda k.\ \mathsf{if}\ t\ k = \emptyset\ \mathsf{then}\ s\ k\ \mathsf{else}\ t\ k \end{array}$$

$$\begin{array}{ll} \_\ \mathsf{hidings}\ \_\ \mathsf{entails}\ \_ & :: (\alpha,\beta)tables \to (\alpha,\gamma)tables \to (\beta \to \gamma \to bool) \to bool \\ t\ \mathsf{hidings}\ s\ \mathsf{entails}\ R & \equiv \forall k.\ \forall x{\in}t\ k.\ \forall y{\in}s\ k.\ R\ x\ y \end{array}$$

2.5.3 Alternatives

An alternative model for partial functions would be a pair of a definedness predicate and a totalized function, but this presentation would be less compact. Yet in some applications, the definedness test is not required. Here the model with a totalized function only would be more convenient than our model where we write $\mathsf{the}\ (t\ x)$ when we know that for the key x an entry must exist. Yet in most applications like type-checking, the definedness test is essential.

Alternative variants of lookup tables are finite sets or lists of pairs. Both are less abstract than partial functions since with them non-uniqueness is possible or even the order of pairs is significant, and lookup is more difficult than the immediate function application $t\ x$. On the other hand, for modeling program semantics the list variant has two important benefits: it is a priori finite, like any part of a program is, and has a rather operational characteristic useful for code generation. This motivates the additional use of type $(\alpha \times \beta)list$ for modeling declarations within a program. This type gives us an implicit finiteness constraint and the possibility for explicit uniqueness check:

$$\begin{array}{l} \mathsf{unique} :: (\alpha \times \beta)list \to bool \\ \mathsf{unique} \equiv \mathsf{nodups} \circ \mathsf{map\ fst} \end{array}$$

For looking up declared entities it is very convenient to transform declaration lists into the abstract tables defined above:

$$\begin{array}{ll} \mathsf{table\_of} & :: (\alpha \times \beta)list \to (\alpha,\beta)table \\ \mathsf{table\_of}\ [] & = \mathsf{empty} \\ \mathsf{table\_of}\ ((k,x)\#t) & = (\mathsf{table\_of}\ t)(k{\mapsto}x) \end{array}$$

This transformation function has the following characteristic properties:

$$\begin{array}{ll} in\_set\_get & \mathsf{unique}\ l \longrightarrow (k,\ x) \in \mathsf{set}\ l \longrightarrow \mathsf{table\_of}\ l\ k = \mathsf{Some}\ x \\ get\_in\_set & \mathsf{table\_of}\ xs\ k = \mathsf{Some}\ y \longrightarrow (k,\ y) \in \mathsf{set}\ xs \end{array}$$

We will use the list variant principally for modeling static declarations while using partial functions for entities derived from these declarations and for the run-time stores within the program state.

2.6 Declarations

In this section we describe our model of Java^{light} programs, which is basically a nested structure of tuples describing classes and their members. It might be worth replacing the tuple representation by (extensible) records. Their explicit naming of fields is more verbose but assists readability (by explicitly relating the values of components with their roles) and simplifies extensions.

2.6.1 Fields and Methods

A *field* declaration [GJS96, §8.3] is indexed by the field name and contains of a modifier and the field type.

$$\begin{aligned} &\mathit{fdecl} = \mathit{ename} \times \mathit{field} \\ &\mathit{field} = \mathit{modi} \times \mathit{ty} \end{aligned}$$

The only field and method *modifier* we currently consider is `static` [Restriction □ ≫ □]. Thus we define

$$\mathit{modi} = \mathit{bool}$$

and projecting on the static modifier is trivial: $\mathsf{static}\ m \equiv m$

A method declaration ($\mathit{sig} \times \mathit{mhead}$ for interfaces and $\mathit{sig} \times \mathit{methd}$ for classes) is indexed by the signature and contains the method head and — if appearing within a class — the method body. The former consists of the modifier, the list of parameter names and the result type, whereas the latter consists of the list of local variables, a statement representing the actual body, and a result expression, explained below. See §6.2.2 for examples.

$$\begin{aligned} &\mathit{mdecl} \ = \mathit{sig} \times \mathit{methd} \\ &\mathit{methd} = \mathit{mhead} \times \mathit{mbody} \\ &\mathit{mhead} = \mathit{modi} \times (\mathit{ename})\mathit{list} \times \mathit{ty} \\ &\mathit{mbody} = (\mathit{ename} \times \mathit{ty})\mathit{list} \times \mathit{stmt} \times \mathit{expr} \end{aligned}$$

Note that *mhead* comprises the information on methods common to both class and interface declarations [Factoring ≫]. For projecting on the return type contained in a method head, we use the abbreviation $\mathsf{mrt}\ mh \equiv \mathsf{snd}\ (\mathsf{snd}\ mh)$.

Like in [DE99], the *result expression* saves us from dealing with `return` statements occurring in arbitrary positions within the method body. Such statements may be replaced by assignments to a suitable result variable followed by a control transfer[5] to the end of the method body, using the result variable as return expression [Reduction ≫]. For `void` methods, we provide a dummy result type (*cf.* §2.2.1) and value (*cf.* §2.3) [Unification ≫].

We do not consider `throws` clauses [Restriction □ ≫ □].

2.6.2 Classes and Interfaces

An *interface* [GJS96, §9.1] declaration is indexed by the interface name and contains a list of superinterface names and the interface body, which is just a list of method declarations. We do not consider interface fields, *i.e.* named constants [Restriction □ ≫ □].

$$\begin{aligned} &\mathit{idecl} = \mathit{tname} \times \mathit{iface} \\ &\mathit{iface} = (\mathit{tname})\mathit{list} \times \mathit{ibody} \\ &\mathit{ibody} = (\mathit{sig} \times \mathit{mhead})\mathit{list} \end{aligned}$$

[5] Some cases like `return` statements within loops would require non-trivial program transformations.

Similarly, a *class* declaration [GJS96, §8.1] is indexed by the class name and specifies the names of the superclass and of implemented interfaces, as well as lists of field and method declarations and a *static initializer* [GJS96, §8.5]. See §6.2.3 for examples. Without loss of expressiveness, we combine all static initializers of a class into a single block of type *stmt* [Reduction]. For every class the superclass entry is explicit [Reduction], and the superclass entry for class `Object` is unused. User-defined *constructors* [GJS96, §8.6] may be emulated using methods called directly after instance creation [Reduction].

$$\begin{aligned}
&cdecl\ = tname \times class\\
&class\ = tname \times (tname)list \times cbody\\
&cbody = (fdecl)list \times (mdecl)list \times stmt
\end{aligned}$$

These type definitions are nested because the inner parts like *cbody* are sometimes used independently of the outer ones [Factoring].

We model the most important language-defined *standard classes*, *viz.* several *standard exceptions* and `Object`. The list of all these classes will be used for examples. We do not give actual definitions for the method declarations contained in them (if any) since they are irrelevant for meta theory [Underspec]. For concrete program verification they can be added on demand, in particular the `equals` method of `Object`.

```
ObjectC ::              cdecl
SXcptC  :: xname → cdecl
Object_mdecls ::  (mdecl)list
SXcpt_mdecls  ::  (mdecl)list

ObjectC    ≡ (Object   ,(arbitrary,[],[],Object_mdecls,Skip)
SXcptC xn ≡ let sc = if xn = Throwable then Object else SXcpt Throwable in
               (SXcpt xn,(sc        ,[],[],SXcpt_mdecls ,Skip))

standard_classes :: (cdecl)list
standard_classes ≡ [ObjectC, SXcptC Throwable,
                    SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
                    SXcptC NegArrSize , SXcptC IndOutBound,  SXcptC ArrStore]
```

Note how we model (through the superclass entry) the fact that the standard exceptions form a flat hierarchy with `Throwable` as its root, which in turn is a direct subclass of `Object`.

2.6.3 Programs

A *program* is a pair of lists of interface and class declarations:

$$prog = (idecl)list \times (cdecl)list$$

We will use the symbol 'Γ' for programs, since programs play the role of the static context in many judgments, *e.g.* typing and evaluation.

There are two access functions on programs, defined as abbreviations:

```
iface :: prog → (tname,iface)table
class :: prog → (tname,class)table

iface Γ I ≡ table_of (fst Γ) I
class Γ C ≡ table_of (snd Γ) C
```

looking up interfaces and classes, respectively. Based on these, we define the predicates

```
is_iface Γ I ≡ iface Γ I ≠ None
is_class Γ C ≡ class Γ C ≠ None
```

testing the existence of an interface or a class, respectively. On top of them, in turn, we define[6] the predicate $\mathsf{is\_type} :: \mathit{prog} \to \mathit{ty} \to \mathit{bool}$ checking for a proper type:

$$
\begin{array}{ll}
\mathsf{is\_type}\ \Gamma\ (\mathsf{PrimT}\ \_) & = \mathsf{True} \\
\mathsf{is\_type}\ \Gamma\ \ \mathsf{NT} & = \mathsf{True} \\
\mathsf{is\_type}\ \Gamma\ (\mathsf{Iface}\ \ I) & = \mathsf{is\_iface}\ \Gamma\ \ I \\
\mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ C) & = \mathsf{is\_class}\ \Gamma\ C \\
\mathsf{is\_type}\ \Gamma\ (T[]) & = \mathsf{is\_type}\ \ \Gamma\ T
\end{array}
$$

2.6.4 Hierarchy Traversal

Later we will need functions for looking up methods and fields by their identifier (*i.e.* signature or field specification $\mathit{fspec} = \mathit{ename} \times \mathit{tname}$, *cf.* §2.8.6) in the context of a given class or interface within a program:

$$
\begin{array}{ll}
\mathsf{imethds} & :: \mathit{prog} \to \mathit{tname} \to (\mathit{sig},\mathit{tname} \times \mathit{mhead})\mathit{tables} \\
\mathsf{cmethd} & :: \mathit{prog} \to \mathit{tname} \to (\mathit{sig},\mathit{tname} \times \mathit{methd})\mathit{table} \\
\mathsf{fields} & :: \mathit{prog} \to \mathit{tname} \to (\mathit{fspec} \times \mathit{field})\mathit{list}
\end{array}
$$

These functions have to take *inheritance* into account, and thus need to traverse the class and interface hierarchies recursively. From the logical perspective this is non-trivial because such a recursive access is well-defined only if the respective subclass or subinterface relation is well-founded. This is an intricate issue easily overlooked or neglected when doing a pen-and-paper formalization, as happened *e.g.* in [DE97a]. After noticing the problem already by careful inspection of their definitions, we took the approach described in this subsection. Syme [Sym99b, §2.1] resorted to inductive definitions instead, thus he does not have to worry about definedness in the first place. Of course, later he has to prove that — under certain conditions — field and method lookup do yield a (unique) result.

Well-Foundedness

In order to state and exploit the well-foundedness of the class and interface hierarchies, we define the notions of a *direct subclass* [GJS96, §8.1.3] and *direct subinterface* [GJS96, §9.1.3] here. We have to do this in anticipation of the type relations introduced in §2.7, because there we unfortunately already need the functions cmethd and imethds. Note that they are expressed with reference to a given program, as will all other type relations.

$$
\begin{array}{l}
\mathsf{subint1} :: \mathit{prog} \to (\mathit{tname} \times \mathit{tname})\mathit{set} \\
\mathsf{subcls1} :: \mathit{prog} \to (\mathit{tname} \times \mathit{tname})\mathit{set}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{subint1}\ \Gamma \equiv \{(I\ ,J\ ).\ \exists i{\in}\mathsf{iface}\ \Gamma\ I\!:\ J{\in}\mathsf{set}\ (\mathsf{fst}\ i)\} \\
\mathsf{subcls1}\ \Gamma \equiv \{(C,D).\ C \neq \mathtt{Object} \wedge (\exists c{\in}\mathsf{class}\ \Gamma\ C\!:\ \mathsf{fst}\ c = D)\}
\end{array}
$$

A binary relation (and its converse) is well-founded if it is finite and acyclic. Finiteness is rather simple here as in any Java program only a finite number of classes and interfaces may exist, and this implicit finiteness constraint carries over to our model since for class and interface declarations we deliberately chose a list representation, which is a priori finite. From the finiteness of its domain and range, one can conclude the finiteness of a relation, and thus we get

$$\mathsf{finite}\ (\mathsf{subint1}\ \Gamma) \ \text{ and } \ \mathsf{finite}\ (\mathsf{subcls1}\ \Gamma)$$

Obtaining acyclicity is a bit more involved. To this end we introduce the notion of *well-structuredness*, which is part of the overall well-formedness notion that we will introduce in

[6] Actually, for technical reasons is_type is defined by mutual primitive recursion with the auxiliary predicate isrtype testing a reference type.

§2.9. Well-structuredness comprises constraints mentioned in [GJS96, §8.1.3, 9.1.3]: A class may not be declared as a subclass of itself, and analogously for interfaces. We model this using predicates on interfaces, classes, and whole programs. For example, an interface is well-structured if none of its direct superinterfaces is at the same time a subinterface of the current interface. It is convenient to require that additionally the superinterfaces actually exist within the program. Analogous conditions have to be met for class declarations, and the well-structuredness for a programs means just well-structuredness of all its classes and interfaces:

$$\begin{array}{lll}
\mathsf{ws_idecl} & :: \ prog \to tname \to (tname)list & \to bool \\
\mathsf{ws_cdecl} & :: \ prog \to tname \to tname & \to bool \\
\mathsf{ws_prog} & :: \ prog \to bool &
\end{array}$$

$$\begin{array}{l}
\mathsf{ws_idecl}\ \Gamma\ I\ si \ \equiv \forall J{\in}\mathsf{set}\ si. \qquad \mathsf{is_iface}\ \Gamma\ J\ \wedge\ (J,I) \notin (\mathsf{subint1}\ \Gamma)^+ \\
\mathsf{ws_cdecl}\ \Gamma\ C\ sc \equiv C \neq \mathtt{Object} \longrightarrow \mathsf{is_class}\ \Gamma\ sc\ \wedge\ (sc,C) \notin (\mathsf{subcls1}\ \Gamma)^+ \\
\mathsf{ws_prog}\ \Gamma \equiv (\forall (I,(si,\ ib)){\in}\mathsf{set}\ (\mathsf{fst}\ \Gamma).\ \mathsf{ws_idecl}\ \Gamma\ I\ si)\ \wedge \\
\qquad\qquad\quad (\forall (C,(sc,cb)){\in}\mathsf{set}\ (\mathsf{snd}\ \Gamma).\ \mathsf{ws_cdecl}\ \Gamma\ C\ sc)
\end{array}$$

Exploiting well-structuredness of programs, we can prove that both the subinterface and the subclass relations are acyclic (and also irreflexive) and thus their converses are well-founded:

$$\mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{acyclic}\ (\mathsf{subint1}\ \Gamma) \quad \text{and} \quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{acyclic}\ (\mathsf{subcls1}\ \Gamma)$$

$$\mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{wf}\ ((\mathsf{subint1}\ \Gamma)^{-1}) \quad \text{and} \quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{wf}\ ((\mathsf{subcls1}\ \Gamma)^{-1})$$

Recursion Operators

Having obtained well-foundedness, we can define the desired lookup functions. In order to factor out the common recursion pattern [Factoring] we define general recursion operators on the interface and class hierarchy: [7]

$$\begin{array}{l}
\mathsf{iface_rec} :: prog \times tname \to \qquad (tname \to ibody \to (\alpha)\mathsf{set} \to \alpha) \to \alpha \\
\mathsf{class_rec} :: prog \times tname \to \alpha \to (tname \to cbody \to \ \alpha \qquad \to \alpha) \to \alpha
\end{array}$$

Next we give their characteristic recursive equations, which can be exploited only if their respective preconditions are met:

$$\begin{array}{ll}
iface_rec & \mathsf{ws_prog}\ \Gamma \longrightarrow \mathsf{iface}\ \Gamma\ I = \mathsf{Some}\ (si,ib) \longrightarrow \\
& \mathsf{iface_rec}\ (\Gamma,I)\ f = f\ I\ ib\ ((\lambda J.\ \mathsf{iface_rec}\ (\Gamma,J)\ f)\ \text{``}\ \mathsf{set}\ si) \\
class_rec & \mathsf{ws_prog}\ \Gamma \longrightarrow \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (sc,si,cb) \longrightarrow \\
& \mathsf{class_rec}\ (\Gamma,C)\ t\ f = f\ C\ cb\ (\mathsf{if}\ C = \mathtt{Object}\ \mathsf{then}\ t\ \mathsf{else}\ \mathsf{class_rec}\ (\Gamma,sc)\ t\ f)
\end{array}$$

The recursion operator for interfaces takes as its argument (next the program and the interface to begin with) a function that combines the current interface and body, as well as the set of entities computed recursively, into a new entity. The operator for classes acts analogously, but does not need to use sets of entities because for any class (except for `Object`, for which we provide an explicit start value) there is exactly one superclass.

Application

Finally, we can define the lookup functions by supplying the recursion operators with suitable arguments:

[7]As a workaround for some limitations of the Isabelle/HOL package for recursive definitions [Sli96], we originally had to introduce an auxiliary well-founded relation, which meanwhile is no longer necessary.

$$\begin{array}{l}
\mathsf{imethds}\ \Gamma\ I \equiv \mathsf{iface_rec}\ (\Gamma{,}I)\\
\qquad\qquad (\lambda I\ ms\ ts.\ (\mathsf{Un_tables}\ ts) \oplus\oplus (\mathsf{o2s} \circ \mathsf{table_of}\ (\mathsf{map}\ (\lambda(s{,}m).\ (s{,}I{,}m))\ ms)))\\
\mathsf{cmethd}\ \ \Gamma\ C \equiv \mathsf{class_rec}\ (\Gamma{,}C)\ \mathsf{empty}\\
\qquad\qquad (\lambda C\ (fs{,}ms{,}ini)\ ts.\ ts \mathbin{++} \mathsf{table_of}\ (\mathsf{map}\ (\lambda(s{,}m).\ (s{,}C{,}m))\ ms))\\
\mathsf{fields}\ \Gamma\ C \equiv \mathsf{class_rec}\ (\Gamma{,}C)\ []\ (\lambda C\ (fs{,}ms{,}ini)\ ts.\ \mathsf{map}\ (\lambda(n{,}t).\ ((n{,}C){,}t))\ fs\ @\ ts)
\end{array}$$

imethds unifies the method tables for the superinterfaces and combines them (implementing inheritance and overriding) with the method declarations of the current interface, which are labeled with the name of this interface. Due to table unification, a possibly non-functional method table is produced, but this does no harm, as discussed in §2.9.2. Analogously, cmethd computes a (unique) table of methods starting from the empty table, and fields collects the list of fields of a class, starting from the empty list. For an example see §6.3.1.

A simple application of cmethd is the predicate

$$\begin{array}{l}
\mathsf{is_methd} :: prog \to tname \to sig \to bool\\
\mathsf{is_methd}\ \Gamma \equiv \lambda C\ sig.\ \mathsf{is_class}\ \Gamma\ C \land \mathsf{cmethd}\ \Gamma\ C\ sig \neq \mathsf{None}
\end{array}$$

Well-foundedness justifies the following rules for induction on the (converse) direct subinterface and subclass relation, which are used for deriving properties of the lookup functions:

$$ws\_subint1\_induct\quad \frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ I \qquad \mathsf{ws\_prog}\ \Gamma\\ \forall I\ is\ ms.\ \mathsf{iface}\ \Gamma\ I = \mathsf{Some}\ (is{,}ms) \land (\forall J \in \mathsf{set}\ is.\\ (I{,}J){\in}\mathsf{subint1}\ \Gamma \land P\ J \land \mathsf{is\_iface}\ \Gamma\ J)) \longrightarrow P\ I\end{array}}{P\ I}$$

$$ws\_subcls1\_induct\quad \frac{\begin{array}{c}\mathsf{is\_class}\ \Gamma\ C \qquad \mathsf{ws\_prog}\ \Gamma\\ \forall C\ D\ c.\ \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (D{,}c) \land (C \neq \mathtt{Object} \longrightarrow\\ (C{,}D){\in}\mathsf{subcls1}\ \Gamma \land P\ D \land \mathsf{is\_class}\ \Gamma\ D)) \longrightarrow P\ C\end{array}}{P\ C}$$

An important property is the relation between the entity (a class or interface) in which a certain member (a method or field) is available and the context where the declaration of the member is actually given and from which it is inherited. For example, for imethds this lemma reads as

$$imethds\_defpl\quad \begin{array}{l}\mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{is\_iface}\ \Gamma\ I \longrightarrow (md{,}mh) \in \mathsf{imethds}\ \Gamma\ I\ sig \longrightarrow\\ (\exists is\ ms.\ \mathsf{iface}\ \Gamma\ md = \mathsf{Some}\ (is,\ ms) \land (I{,}md) \in (\mathsf{subint1}\ \Gamma)^* \land\\ \qquad\qquad \mathsf{table\_of}\ ms\ sig = \mathsf{Some}\ mh) \land (md{,}mh) \in \mathsf{imethds}\ \Gamma\ md\ sig\end{array}$$

meaning that for a well-formed program, if an interface has a certain method as its member, then the defining interface is a superinterface and directly contains the method declaration that also has been found in the subinterface. Of course, there are analogous lemmas for the other lookup functions.

We will further need a lemma concerning uniqueness of fields:

$$ws\_unique\_fields\quad \begin{array}{l}\mathsf{ws\_prog}\ \Gamma \land \mathsf{is\_class}\ \Gamma\ C \land\\ (\forall C\ D\ s\ fs\ r.\ \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (D{,}s{,}fs{,}r) \longrightarrow \mathsf{unique}\ fs) \longrightarrow\\ \mathsf{unique}\ (\mathsf{fields}\ \Gamma\ C)\end{array}$$

2.7 Type Relations

In the Java specification, most type relations are called *conversions* capturing the idea that a value of one type may be transformed into a value of another type. This is adequate for many primitive types, but not really for reference types, since reference values never change even when being interpreted as belonging to different types. As the only conversion applicable on the primitive types that we consider is the (trivial) identity conversion, we do not encounter any transformation of values but just the relational aspect on types.

2.7.1 Basic Relations

Two of the three basic relations on reference types, the direct subinterface and subclass relations, have already been defined in §2.6.4. Here we just provide mixfix syntax for them and give the connection

$$\Gamma \vdash I \prec_{\mathsf{i}}^{1} J \;\equiv (I,\ J) \in \mathsf{subint1}\ \Gamma$$
$$\Gamma \vdash C \prec_{\mathsf{c}}^{1} D \equiv (C,D) \in \mathsf{subcls1}\ \Gamma$$

The third one is the *direct implementation* [GJS96, §8.1.3] relation, which we give the syntax

$$prog \vdash tname \leadsto^{1} tname$$

and define as

$$implmt\mathsf{1}\ \Gamma \equiv \{(C,I).\ C \neq \texttt{Object} \wedge (\exists c{\in}\mathsf{class}\ \Gamma\ C\text{: } I \in \mathsf{set}\ (\mathsf{fst}\ (\mathsf{snd}\ c)))\}$$

Note that for all three relations, it easily follows from their definitions that the left of both type arguments is a proper type, *i.e.* is_class or is_iface holds for it, respectively, which is essential to guarantee the finiteness of the relations. One could consider requiring (for symmetry) that the right type argument is also proper, but this complicates matters unnecessarily.

2.7.2 Transitive Closures

All three basic relations are extended by some kind of transitive closure. The *subinterface* [GJS96, §9.1.3] and *subclass* [GJS96, §8.1.3] relations

$$prog \vdash tname \preceq_{\mathsf{i}} tname$$
$$prog \vdash tname \preceq_{\mathsf{c}} tname$$

are originally defined as the transitive closure of their direct counterparts. We deviate from this for convenience by taking also the reflexive closures:

$$\Gamma \vdash I \;\preceq_{\mathsf{i}} J \;\equiv (I,\ J) \in (\mathsf{subint1}\ \Gamma)^{*}$$
$$\Gamma \vdash C \;\preceq_{\mathsf{c}} D \equiv (C,D) \in (\mathsf{subcls1}\ \Gamma)^{*}$$

The *implementation* relation

$$prog \vdash tname \leadsto tname$$

combines its direct counterpart with the subinterface and subclass relations. We exactly mirror its specification [GJS96, §8.1.4], which is in fact an inductive definition:

$$direct\ \frac{\Gamma \vdash C \leadsto^{1} J}{\Gamma \vdash C \leadsto J} \qquad subint\ \frac{\Gamma \vdash C \leadsto^{1} I \quad \Gamma \vdash I \preceq_{\mathsf{i}} J}{\Gamma \vdash C \leadsto J} \qquad subcls1\ \frac{\Gamma \vdash C \prec_{\mathsf{c}}^{1} D \quad \Gamma \vdash D \leadsto J}{\Gamma \vdash C \leadsto J}$$

Inductive definitions are a very flexible and powerful mechanism that we use heavily for defining various kinds of relations and judgments.

We derive, typically by induction, a number of properties, for example

$$\textit{subcls\_implmt} \quad \Gamma \vdash C \preceq_{\mathsf{c}} D \wedge \Gamma \vdash D \leadsto I \longrightarrow \Gamma \vdash C \leadsto I$$

and

$$\textit{subcls\_ObjectI} \quad \mathsf{ws\_prog}\ \Gamma \wedge \mathsf{is\_class}\ \Gamma\ C \longrightarrow \Gamma \vdash C \preceq_{\mathsf{c}} \mathtt{Object}$$

2.7.3 Widening

The most important type relation is *widening* [GJS96, §5.3]:

$prog \vdash ty \preceq ty$

where we combine the identity conversion and the widening reference conversion. $\Gamma \vdash S \preceq T$ means that S is a syntactic subtype of T, *i.e.* in any expression context (for instance assignments and method invocations) expecting a value of type T, a value of type S may occur. Note that this does not necessarily mean that type S semantically behaves like type T, but only that it offers at least a syntactically compatible set of fields and methods. The widening relation is defined inductively as

$$\textit{refl}\ \frac{}{\Gamma \vdash T \preceq T} \qquad \textit{null}\ \frac{}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

$$\textit{subint}\ \frac{\Gamma \vdash I \preceq_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \textit{int\_obj}\ \frac{}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathtt{Object}}$$

$$\textit{subcls}\ \frac{\Gamma \vdash C \preceq_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \textit{implmt}\ \frac{\Gamma \vdash C \leadsto I}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I}$$

$$\textit{array}\ \frac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash \mathsf{RefT}\ S[] \preceq \mathsf{RefT}\ T[]} \qquad \textit{arr\_obj}\ \frac{}{\Gamma \vdash T[] \preceq \mathsf{Class}\ \mathtt{Object}}$$

The most important property of the widening relation is transitivity:

$$\textit{ws\_widen\_trans} \quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \Gamma \vdash S \preceq U \longrightarrow \Gamma \vdash U \preceq T \longrightarrow \Gamma \vdash S \preceq T$$

We prove this by rule induction where we need well-structuredness for applying the lemma *subcls_ObjectI* given above. Widening is also antisymmetric and enjoys many other (simple) properties like

$$\textit{widen\_Class\_Iface\_eq} \quad \Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I = \Gamma \vdash C \leadsto I$$

The widening relation carries over canonically to lists of types:

$\Gamma \vdash Ts[\preceq] Ts' \equiv \mathsf{list\_all2}\ (\lambda T\ T'.\ \Gamma \vdash T \preceq T')\ Ts\ Ts'$
where
$\mathsf{list\_all2} :: (\alpha \to \beta \to \mathit{bool}) \to (\alpha)\mathit{list} \to (\beta)\mathit{list} \to \mathit{bool}$
$\mathsf{list\_all2}\ P\ xs\ ys \equiv \mathsf{length}\ xs = \mathsf{length}\ ys \wedge (\forall (x,y) \in \mathsf{set}\ (\mathsf{zip}\ xs\ ys).\ P\ x\ y)$

2.7.4 Narrowing and Casting

The *narrowing* [GJS96, §5.1.5] relation

$prog \vdash ty \succ ty$

is a kind of converse of the widening relation, where $\Gamma \vdash S \succ T$ means that an element of (a subtype of) S might be an element of T, but in a way that cannot be guaranteed statically. An explicit inductive definition of a relation fulfilling these constraints while making the (approximate) static check as strict as possible would be even more involved than the one actually given in the specification. The definition given there — which we follow except for a slight generalization concerning the *subint* case as noted below — is actually too permissive, which could be fixed using additional side conditions.

$$subcls\ \frac{\Gamma \vdash C \preceq_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ D \succ \mathsf{Class}\ C} \qquad int\_cls\ \frac{}{\Gamma \vdash \mathsf{Iface}\ I \succ \mathsf{Class}\ C} \qquad implmt\ \frac{\neg \Gamma \vdash C \rightsquigarrow I}{\Gamma \vdash \mathsf{Class}\ C \succ \mathsf{Iface}\ I}$$

$$array\ \frac{\Gamma \vdash \mathsf{RefT}\ S \succ \mathsf{RefT}\ T}{\Gamma \vdash \mathsf{RefT}\ S[] \succ \mathsf{RefT}\ T[]} \qquad obj\_arr\ \frac{}{\Gamma \vdash \mathsf{Class}\ \mathtt{Object} \succ T[]}$$

$$subint\ \frac{\begin{array}{c}\neg \Gamma \vdash I \preceq_{\mathsf{i}} J \\ (\mathsf{imethds}\ \Gamma\ I)\ \mathsf{hidings}\ (\mathsf{imethds}\ \Gamma\ J)\ \mathsf{entails}\ (\lambda(\_,\ m)\ (\_, m').\ \Gamma \vdash \mathsf{mrt}\ m \preceq \mathsf{mrt}\ m')\end{array}}{\Gamma \vdash \mathsf{Iface}\ I \succ \mathsf{Iface}\ J}$$

The *subint* case contains the side condition that for any pair of methods (one of interface I, the other of J) with a common signature the result types are in widening relation, while [GJS96, §5.1.5] demands equality [Generalization □ ⋙ □]. On the other hand, corresponding conditions on the result types of methods should be added to the first three cases as well [Clarification □ ⋙ □] and thus the narrowing relation could be strengthened avoiding some cases of type casts that are hopeless anyway (in the sense that the cast will always fail). In earlier versions of our formalization we did this partially, but this was not worth the effort because it does not have any effect on the properties of narrowing we actually need.

Unification of widening and narrowing leads to the *casting* [GJS96, §5.5] relation

$prog \vdash ty \preceq ?\ ty$

where $\Gamma \vdash S \preceq ?\ T$ means that a value of type S can be cast to type T, *i.e.* at least possibly conforms to T.

$$widen\ \frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq ?\ T} \qquad narrow\ \frac{\Gamma \vdash S \succ T}{\Gamma \vdash S \preceq ?\ T}$$

Surprisingly, the only properties of narrowing and casting we actually need are

$$\begin{array}{lll} cast\_RefT2 & \Gamma \vdash S \preceq ?\ \mathsf{RefT}\ R & \longrightarrow \exists t.\ S = \mathsf{RefT}\ t \\ cast\_PrimT2 & \Gamma \vdash S \preceq ?\ \mathsf{PrimT}\ pt & \longrightarrow \exists t.\ S = \mathsf{PrimT}\ t \wedge \Gamma \vdash \mathsf{PrimT}\ t \preceq \mathsf{PrimT}\ pt \end{array}$$

where narrowing is only indirectly involved. Thus one could dispense with the two inductive definitions altogether and require just these two properties as axioms [Underspec □ ⋙ □].

2.8 Well-Typedness

Classically, the notion of *well-typedness* serves as a sanity check on the possible outcome of expressions. It extends naturally to variables and enclosing statements. Since typing in Java is unique (in particular, there is no subsumption, which would defeat the method overloading mechanism), typing rules are used also for inferring the types of expressions and variables. We further use them for expressing related well-formedness conditions — namely the existence of variables, fields and methods —, and for computing type annotations (*cf.* §2.4.2).

2.8.1 Environments

Typing is always done relative to some environment. A *static environment* consists of a global part, namely the program Γ, and a *local environment* (typically denoted by 'Λ') that gives the types of the current local variables including the This pointer (in non-static methods) and method parameters. Recall that we do not consider nested blocks. The terms prg and lcl serve as projection operators.

$$lenv = (lname,\ ty)table$$
$$env = prog\ \times\ lenv$$

$$\mathsf{prg}\ (\Gamma,\Lambda) \equiv \Gamma$$
$$\mathsf{lcl}\ \ (\Gamma,\Lambda) \equiv \Lambda$$

When defining a transition semantics and proving type soundness for it, as will be discussed in §4.6.2, one would have to assign types to partially evaluated terms. This requires a *dynamic type environment*, *viz.* the type dyn_ty already introduced in §2.3 which assigns types to references.

2.8.2 Judgments

As motivated in §2.4.4, we unify all kinds of typing judgments into one, which will considerably reduce redundancy in their applications, in particular when expressing type safety (*cf.* §4.4). Concerning the type, we can combine expressions, variables and statements since they have a single type if we consider statements to have the dummy type PrimT `void`. Only expression lists, of course, are associated with lists of types.

$$tys = ty\ +\ (ty)list$$

In order to provide for a transition semantics,[8] we have extended the well-typedness judgment and the typing rules to include also the dynamic environment, whereas in the judgments that we actually use later this parameter is set to $\mathsf{empty\_dt} \equiv \lambda a.\ \mathsf{None}$.

The extended judgment has the following form.[9]

$$env,\mathsf{dyn\_ty}\models term::tys$$

We define also syntactic variants for the four kinds of terms, hiding the necessary injections. Note that the count of dashes behind the '::' symbol below is mnemonic for the number of values produced: a statement yields no, an expression one, a variable two (*cf.* §3.2.10), and an expression list any number of values.

[8] We planned originally to give an additional transition semantics, but this has been suspended.

[9] As required for technical reasons by the definition package for inductive relations, the internal form is $\mathsf{wt} :: (env \times \mathsf{dyn\_ty} \times term \times tys)set$. This is translated to the more readable external form via $E,dt\models t::T \equiv (E,dt,t,T) \in \mathsf{wt}$. Here the *env* and dyn_ty parts are not just parameters of wt (which would result in the type $env \to \mathsf{dyn\_ty} \to (term \times tys)set$) because the environment changes within the typing rule for the `try _ catch _` statement.

$env,\mathsf{dyn\_ty}\models stmt::\surd$
$env,\mathsf{dyn\_ty}\models expr::-ty$
$env,\mathsf{dyn\_ty}\models var\ ::=ty$
$env,\mathsf{dyn\_ty}\models (expr)\,list::\doteq(ty)\,list$

$E,dt\models s::\surd \quad\equiv\quad E,dt\models \mathsf{In1r}\ s::\mathsf{Inl}\ (\mathsf{PrimT}\ \texttt{void})$
$E,dt\models e::-T \ \equiv\ E,dt\models \mathsf{In1l}\ e::\mathsf{Inl}\ T$
$E,dt\models e::=T \ \equiv\ E,dt\models \mathsf{In2}\ \ e::\mathsf{Inl}\ T$
$E,dt\models e::\doteq T \ \equiv\ E,dt\models \mathsf{In3}\ \ e::\mathsf{Inr}\ T$

Replacing '$\models$' by '$\vdash$', we further introduce the following specialized typing judgments that we will actually use in the proof of type safety for evaluation semantics and in the example given in §6.

$E\vdash t::\ \ T \equiv E,\mathsf{empty\_dt}\models t::\ T$

$E\vdash s::\surd \quad \equiv E,\mathsf{empty\_dt}\models s::\surd$
$E\vdash e::-T \equiv E,\mathsf{empty\_dt}\models e::-T$
$E\vdash e::=T \equiv E,\mathsf{empty\_dt}\models e::=T$
$E\vdash e::\doteq T \equiv E,\mathsf{empty\_dt}\models e::\doteq T$

The typing rules, given next, are modeled conveniently as inductive definitions. See §6.3.1 for application examples.

2.8.3 Statements

The type-checking rules for the standard statements [GJS96, §14] are standard:

$$\frac{}{E,dt\models \mathsf{Skip}::\surd} \qquad \frac{E,dt\models e::-T}{E,dt\models \mathsf{Expr}\ e::\surd} \qquad \frac{E,dt\models e::-\mathsf{PrimT}\ \texttt{boolean} \quad E,dt\models c::\surd}{E,dt\models \texttt{while}(e)\ c::\surd}$$

$$\frac{E,dt\models c_1::\surd \quad E,dt\models c_2::\surd}{E,dt\models c_1\,\texttt{;}\ c_2::\surd} \qquad \frac{E,dt\models e::-\mathsf{PrimT}\ \texttt{boolean} \quad E,dt\models c_1::\surd \quad E,dt\models c_2::\surd}{E,dt\models \texttt{if}(e)\ c_1\ \texttt{else}\ c_2::\surd}$$

Also the rules for most of the Java-specific statements are straightforward. Just note the use of the subclass relation in two of the following rules ensuring that a value thrown or caught as an exception is indeed an exception object.

$$\frac{E,dt\models e::-\mathsf{Class}\ tn \quad \mathsf{prg}\ E\vdash tn\preceq_{\mathsf{c}}\mathsf{SXcpt}\ \texttt{Throwable}}{E,dt\models \texttt{throw}\ e::\surd} \qquad \frac{E,dt\models c_1::\surd \quad E,dt\models c_2::\surd}{E,dt\models c_1\ \texttt{finally}\ c_2}$$

$$\frac{\begin{array}{c}(\Gamma,\Lambda),dt\models c_1::\surd \quad \Gamma\vdash tn\preceq_{\mathsf{c}}\mathsf{SXcpt}\ \texttt{Throwable}\\ \Lambda\ (\mathsf{EName}\ vn)=\mathsf{None} \quad (\Gamma,\Lambda[\mathsf{EName}\ vn\mapsto\mathsf{Class}\ tn]),dt\models c_2::\surd\end{array}}{(\Gamma,\Lambda),dt\models \texttt{try}\ c_1\ \texttt{catch}(tn\ vn)\ c_2::\surd} \qquad \frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ C}{E,dt\models \mathsf{init}\ C::\surd}$$

The `try _ catch _` statement is the only one that involves a change of the type environment, namely to include typing information for the exception parameter. The name of this parameter is required to be new in the local environment.

2.8.4 Expressions

The first few of the typing rules for expressions [GJS96, §15] straightforwardly follow the specification.

$$\frac{\mathsf{typeof}\ dt\ x=\mathsf{Some}\ T}{E,dt\models \mathsf{Lit}\ x::-T} \qquad \frac{\begin{array}{c}E,dt\models e_0::-\mathsf{PrimT}\ \texttt{boolean} \quad E,dt\models e_1::-T_1 \quad E,dt\models e_2::-T_2\\ \mathsf{prg}\ E\vdash T_1\preceq T_2\ \wedge\ T=T_2\ \ \vee\ \ \mathsf{prg}\ E\vdash T_2\preceq T_1\ \wedge\ T=T_1\end{array}}{E,dt\models e_0\ \texttt{?}\ e_1\ \texttt{:}\ e_2::-T}$$

$$\frac{\textsf{is\_class}\ (\textsf{prg}\ E)\ C}{E,dt\models \texttt{new}\ C{::}{-}\textsf{Class}\ C} \qquad \frac{\textsf{is\_type}\ (\textsf{prg}\ E)\ T \quad E,dt\models i{::}{-}\textsf{PrimT}\ \texttt{int}}{E,dt\models NewA\ T[i]{::}{-}T[]}$$

$$\frac{E,dt\models e{::}{-}T \quad \textsf{prg}\ E,dt\models T\preceq?\ T' \quad \textsf{is\_type}\ (\textsf{prg}\ E)\ T'}{E,dt\models \textsf{Cast}\ T'\ e{::}{-}T'}$$

$$\frac{E,dt\models e{::}{-}\textsf{RefT}\ T \quad \textsf{prg}\ E,dt\models \textsf{RefT}\ T\preceq?\ \textsf{RefT}\ T'}{E,dt\models e\ \texttt{instanceof}\ T'{::}{-}\textsf{PrimT}\ \texttt{boolean}}$$

When using the rule for literal values Lit with dt set to empty_dt, addresses are prohibited as literal values, which is what we need for an evaluation semantics regarding terms as static entities only.

Variable access includes reading access to the This pointer. Since on the contrary, assigning to This is prohibited, variable assignment rules out this special case.

$$\frac{E,dt\models va{::}{=}T}{E,dt\models \textsf{Acc}\ va{::}{-}T} \qquad \frac{E,dt\models va{::}{=}T \quad va \neq \textsf{LVar This} \quad E,dt\models v\ {::}{-}T' \quad \textsf{prg}\ E\vdash T'\preceq T}{E,dt\models va\texttt{:=}v{::}{-}T'}$$

Note that in our model the type of an assignment is determined by the right-hand side (as opposed to the left-hand side) [Generalization]. Thus less type information is lost and more programs are rendered legal. When an assignment is used as an argument to a method call, a type cast to the type of the left-hand side may be used to avoid a potentially different resolution of method overloading.

The rule for `super` [GJS96, §15.10.2, 15.11.1] checks if This is accessible, and unless the corresponding class is `Object`, returns the superclass of that class.

$$\frac{\textsf{lcl}\ E\ \textsf{This} = \textsf{Some}\ (\textsf{Class}\ C) \quad C \neq \texttt{Object} \quad \textsf{class}\ (\textsf{prg}\ E)\ C\ = \textsf{Some}\ (D,\ \_)}{E,dt\models \texttt{super}{::}{-}\textsf{Class}\ D}$$

2.8.5 Methods

The most complex expressions, also concerning their typing, are method calls [GJS96, §15.11], so they deserve special attention. See also §6.3.2 for a running example of the definitions given below.

Method Calls

A method call $\{\_,\_,\_\}e..mn(\{\_\}ps)$ is type-correct if the expression e has some reference type t, the parameters ps have some types pTs, and there is a unique "maximally specific" method (see below) with signature (mn,pTs) accessible from t:

$$\frac{E,dt\models e{::}{-}\textsf{RefT}\ t \quad E,dt\models ps{::}{-}pTs \quad \textsf{max\_spec}\ (\textsf{prg}\ E)\ t\ (mn,pTs) = \{((md,(m,pns,rT)),pTs')\}}{E,dt\models \{t,md,\textsf{invmode}\ (\textsf{static}\ m)\ e\}e..mn(\{pTs'\}ps){::}{-}rT}$$

In this case, the method call is annotated with t, the defining class md of the method found, its parameter types pTs', and its invocation mode, as computed by the auxiliary function

invmode :: $modi \rightarrow expr \rightarrow inv\_mode$
invmode $m\ e \equiv$ if static m then Static else if $e = \texttt{super}$ then Super else IntVir

The annotation t is not needed for the operational semantics, but will be used in the axiomatic semantics, whereas md serves as the class used for static method calls. The annotation pTs' resolves static overloading and thus will be used — in conjunction with the method name — for method lookup at run time.

Resolution of Static Overloading

According to [GJS96, §15.11.2], static overloading is resolved by selecting the most specific method applicable (if exists). To model this, we use the following auxiliary functions and relations. For a given program, mheads returns the set of all extended method heads with signature *sig* available for a reference type *rt*, where by method head *emhead* we mean the pair of the defining class or interface and the method head.

The function mheads relies on $\mathsf{cmheads}$ dealing with the special case of classes. Note that an interface may contain more than one method with a given signature. Furthermore, methods of class `Object` are accessible for any interface ([Correction ≫ □], a omission of [GJS96] first reported by [PB97]) or array type [GJS96, §10.7].

$$
\begin{array}{l}
\mathit{emhead} = \mathit{ref\_ty} \times \mathit{mhead} \\
\mathsf{cmheads} :: \mathit{prog} \to \mathit{tname} \to \mathit{sig} \to \mathit{emhead\ set} \\
\mathsf{mheads}\ \ :: \mathit{prog} \to \mathit{ref\_ty} \to \mathit{sig} \to \mathit{emhead\ set}
\end{array}
$$

$$
\mathsf{cmheads}\ \Gamma\ C \equiv \lambda \mathit{sig}.\ (\lambda(C,(h,b)).\ (\mathsf{ClassT}\ \ C,h))\ \text{``}\ \mathsf{o2s}\ (\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig})
$$

$$
\begin{array}{l}
\mathsf{mheads}\ \Gamma\ \ \ \mathsf{NullT}\ \ \ \ \ = \lambda \mathit{sig}.\ \emptyset \\
\mathsf{mheads}\ \Gamma\ (\mathsf{IfaceT}\ \ I) = \lambda \mathit{sig}.\ (\lambda(I,h).(\mathsf{IfaceT}\ \ I,h))\ \text{``}\ \mathsf{imethds}\ \Gamma\ I\ \mathit{sig} \cup \mathsf{cmheads}\ \Gamma\ \mathtt{Object}\ \mathit{sig} \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ClassT}\ \ C) = \mathsf{cmheads}\ \Gamma\ C \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ArrayT}\ \ T) = \mathsf{cmheads}\ \Gamma\ \mathtt{Object}
\end{array}
$$

The set of methods of a reference type applicable for a given signature consists of those methods returned by mheads whose parameter types are supertypes (by widening) of the corresponding types in the signature:

$$
\begin{array}{l}
\mathsf{appl\_methds}\ :: \mathit{prog} \to\ \ \mathit{ref\_ty} \to \mathit{sig} \to (\mathit{emhead} \times (\mathit{ty})\mathit{list})\ \ \mathit{set} \\
\mathsf{appl\_methds}\ \Gamma\ \mathit{rt}\ (\mathit{mn},\mathit{Ts}) \equiv \{(\mathit{mh},\mathit{Ts}')\ |\mathit{mh}\ \mathit{Ts}'.\ \mathit{mh} \in \mathsf{mheads}\ \Gamma\ \mathit{rt}\ (\mathit{mn},\mathit{Ts}') \wedge \Gamma \vdash \mathit{Ts}[\preceq]\mathit{Ts}'\}
\end{array}
$$

A method is *more specific* than another iff the lists of parameter types and[10] the defining entities (*i.e.* classes or interfaces) are in widening relation:

$$
\begin{array}{l}
\mathsf{more\_spec} :: \mathit{prog} \to \mathit{emhead} \times (\mathit{ty})\mathit{list} \to \mathit{emhead} \times (\mathit{ty})\mathit{list} \to \mathit{bool} \\
\mathsf{more\_spec}\ \Gamma\ ((\mathit{md},\mathit{mh}),\mathit{pTs})\ ((\mathit{md}',\mathit{mh}'),\mathit{pTs}') \equiv \Gamma \vdash \mathit{pTs}[\preceq]\mathit{pTs}' \wedge \Gamma \vdash \mathsf{RefT}\ \ \mathit{md} \preceq \mathsf{RefT}\ \ \mathit{md}'
\end{array}
$$

The maximally specific methods are the applicable methods for which no more specific applicable method exists:

$$
\begin{array}{l}
\mathsf{max\_spec} :: \mathit{prog} \to\ \ \mathit{ref\_ty} \to \mathit{sig} \to (\mathit{emhead} \times (\mathit{ty})\mathit{list})\ \ \mathit{set} \\
\mathsf{max\_spec}\ \Gamma\ \mathit{rt}\ \mathit{sig} \equiv \{m|\ m \in \mathsf{appl\_methds}\ \Gamma\ \mathit{rt}\ \mathit{sig}\ \wedge \\
\qquad\qquad\qquad\qquad (\forall m' \in \mathsf{appl\_methds}\ \Gamma\ \mathit{rt}\ \mathit{sig}.\ \mathsf{more\_spec}\ \Gamma\ m'\ \ m \longrightarrow m' = m)\}
\end{array}
$$

If $\mathsf{max\_spec}$ yields only one method, it is the *most specific* one.

As observed by [AZD00], the relation behind $\mathsf{max\_spec}$ can be relaxed by leaving out the comparison of the defining entity. This extends the set of most specific methods and therefore renders more programs legal [Generalization ≫ □], without altering the operational semantics. Taking into account the defining entity is doubtful from the perspective of software engineering anyway. Thus the version of $\mathsf{more\_spec}$ we actually use meanwhile is

$$
\mathsf{more\_spec}\ \Gamma\ (\mathit{mh},\mathit{pTs})\ (\mathit{mh}',\mathit{pTs}') \equiv \Gamma \vdash \mathit{pTs}[\preceq]\mathit{pTs}'
$$

Moreover, for type soundness the only property of $\mathsf{more\_spec}$, $\mathsf{appl\_methds}$ and $\mathsf{max\_spec}$ actually required is

$$
(\mathit{mh},\mathit{pTs}') \in \mathsf{max\_spec}\ \Gamma\ \ T\ (\mathit{mn},\mathit{pTs}) \longrightarrow \mathit{mh} \in \mathsf{mheads}\ \Gamma\ \ T\ (\mathit{mn},\mathit{pTs}') \wedge \Gamma \vdash \mathit{pTs}[\preceq]\mathit{pTs}'
$$

such that we could leave out their definitions entirely, just declare the constant $\mathsf{max\_spec}$, and assert this property [Underspec ≫ □].

[10] See also the modified version below.

Method Implementations

A method implementation $\mathsf{Methd}\ C\ sig$ has a type T if C is a proper class and contains a method with signature sig whose body is of type T. This in turn means that the defining entity is a proper class, the statement block is well-typed and the result has type T.

$$\frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ C \quad \mathsf{cmethd}\ (\mathsf{prg}\ E)\ C\ sig = \mathsf{Some}\ (md,\_,\_,blk,res) \quad E,dt\models \mathsf{Body}\ md\ blk\ res{::}{-}T}{E,dt\models \mathsf{Methd}\ C\ sig{::}{-}T}$$

$$\frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ D \quad E,dt\models blk{::}\surd \quad E,dt\models res{::}{-}T}{E,dt\models \mathsf{Body}\ D\ blk\ res{::}{-}T}$$

2.8.6 Variables

The rule for local variables [GJS96, §15.13.1] uses the local environment to check if the variable exists and to look up its type:

$$\frac{\mathsf{lcl}\ E\ vn = \mathsf{Some}\ T \quad \mathsf{is\_type}\ (\mathsf{prg}\ E)\ T}{E,dt\models \mathsf{LVar}\ vn{::}{=}T}$$

We require the extra condition that the type is proper because otherwise we would have to introduce and use the well-formedness condition $\forall vn.\ \forall T{\in}\mathsf{lcl}\ E\ vn\colon \mathsf{is\_type}\ (\mathsf{prg}\ E)\ T$ for the local part of environments. We could derive this well-formedness property for environments obtained from well-formed method declarations, but we want to use the typing rules also for terms where the connection to method declarations is not given.

The typing of array variables [GJS96, §15.12] is straightforward, whereas the typing of field variables [GJS96, §15.10.1] is more sophisticated, as described below.

$$\frac{E,dt\models e{::}{-}T[\,] \quad E,dt\models i{::}{-}\mathsf{PrimT}\ \mathtt{int}}{E,dt\models e[i]{::}{=}T} \qquad \frac{E,dt\models e{::}{-}\mathsf{Class}\ C \quad \mathsf{cfield}\ (\mathsf{prg}\ E)\ C\ fn = \mathsf{Some}\ (fd,(m,fT))}{E,dt\models \{fd,\mathsf{static}\ m\}e..fn{::}{=}fT}$$

The function

$\mathsf{cfield} :: prog \to tname \to (ename,\ tname \times field)table$

is a variant of fields, defined as

$\mathsf{cfield}\ \Gamma\ C \equiv \mathsf{table\_of}\ ((\mathsf{map}\ (\lambda((fn,fd),T).\ (fn,(fd,T))))\ (\mathsf{fields}\ \Gamma\ C))$

It implements a field lookup based on the field name alone, *i.e.* hiding of fields (of different classes) with equal names. A field access $\{\_,\_\}e..fn$ is annotated with the defining class fd and the modifier `static` of the field found by searching the class hierarchy for the name fn (starting from the type $\mathsf{Class}\ C$ of the reference expression e). The annotation fd will be used at run time to access the field (in the context of possibly a subclass of C) by calling table_of fields with the pair (fn,fd) as its search key argument. Combining the field name and the defining class, as given by the type

$fspec = ename \times tname$

is necessary to access the field because in the new context it is possibly hidden. Thus static binding for fields is ensured.

2.8.7 Expression Lists

Expression lists are used as arguments for method calls. As their typing is canonical, it is not even explicitly specified in [GJS96].

$$\frac{}{E,dt\models [\,]{::}\dot{=}[\,]} \qquad \frac{E,dt\models e{::}{-}T \quad E,dt\models es{::}\dot{=}Ts}{E,dt\models e\#es{::}\dot{=}T\#Ts}$$

2.8.8 Properties

Unifying the different kinds of typing judgments — in the absence of dependent typing for HOL — brings a slight technical complication: we need the fact that the injections used for the term argument of judgments $E,dt \models t :: U$ match the injections used for the type argument:

$$\begin{array}{l} E,dt \models t :: U \longrightarrow \mathsf{case}\ t\ \mathsf{of}\ \mathsf{In1}\ ec \to (\mathsf{case}\ ec\ \mathsf{of}\ \mathsf{Inl}\ e \to \exists T.\ U = \mathsf{Inl}\ T \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{Inr}\ c \to \qquad U = \mathsf{Inl}\ (\mathsf{PrimT}\ \mathtt{void})) \\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{In2}\ v \to (\exists T.\ \ U = \mathsf{Inl}\ T\ ) \\ \qquad\qquad\qquad\qquad\qquad \mid \mathsf{In3}\ es \to (\exists Ts.\ U = \mathsf{Inr}\ Ts) \end{array}$$

This is proved easily by case distinction: any expression or variable is associated with a single type, any statement with the dummy type `void`, and any expression list with a list of types.

For well-structured programs (which are needed to ensure anti-symmetry of the widening relation), the typing of terms is unique:

$$\mathsf{ws\_prog}\ (\mathsf{fst}\ E) \wedge E,dt \models t :: T \wedge E,dt \models t :: T' \longrightarrow T = T'$$

This property can be shown straightforwardly by rule induction, yet is not actually used.

2.9 Well-Formedness

Complementing the well-typedness of terms, we define the notion of *well-formedness* for classes, interfaces, their members, and whole programs. It reflects the static checks done by the compiler for global sanity of all declarations. The well-formedness conditions are naturally expressed as a hierarchy of predicates reflecting the structure of declarations. For an application example see §6.3.1. With a very powerful type system including predicate subtyping combined with mutual recursion, they could be incorporated directly into the model of declarations, but this is by no means necessary.

2.9.1 Fields and Methods

A field declaration is well-formed iff the type of the field exists [GJS96, §8.3]:

$$\begin{array}{l} \mathsf{wf\_fdecl}\ :: prog \to fdecl \to bool \\ \mathsf{wf\_fdecl}\ \Gamma\ (fn,(m,ft)) \equiv \mathsf{is\_type}\ \Gamma\ ft \end{array}$$

A method declaration is well-formed only if its head is well-formed [GJS96, §8.4], which means that the numbers of parameter names and types agree, all parameter and result types exist, and the parameter names are distinct.

$$\begin{array}{l} \mathsf{wf\_mhead} :: prog \to sig \to mhead \to bool \\ \mathsf{wf\_mhead}\ \Gamma\ (mn,pTs)\ (m,pns,rT) \equiv \mathsf{length}\ pTs = \mathsf{length}\ pns\ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\forall T \in \mathsf{set}\ pTs.\ \mathsf{is\_type}\ \Gamma\ T) \wedge \mathsf{is\_type}\ \Gamma\ rT\ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{nodups}\ pns \end{array}$$

If the method declaration appears in a class (given as an extra parameter to wf_mdecl), there is also a method body, and therefore several additional conditions have to be met. If the defining class is `Object`, the method must not be static. The names of the local variables [GJS96, §14.3] must be unique and may not hide the parameters, and all types of the local variables must exist. The method body has to be well-typed (in the static context of its parameter types, local variables, and the current class unless the method is static), and its result expression must have a type that widens to the result type.

$$
\begin{array}{l}
\mathsf{wf\_mdecl}\ ::\ \mathit{prog} \to \mathit{tname} \to \mathit{mdecl} \to \mathit{bool}\\
\mathsf{wf\_mdecl}\ \Gamma\ C\ ((\mathit{mn},\mathit{pTs}),(\mathit{m},\mathit{pns},\mathit{rT}),\mathit{lvars},\mathit{blk},\mathit{res}) \equiv \mathsf{wf\_mhead}\ \Gamma\ (\mathit{mn},\mathit{pTs})\ (\mathit{m},\mathit{pns},\mathit{rT}) \wedge\\
\quad\quad (C = \texttt{Object} \longrightarrow \neg\mathsf{static}\ m) \wedge\\
\quad\quad \mathsf{unique}\ \mathit{lvars} \wedge (\forall \mathit{pn}{\in}\mathsf{set}\ \mathit{pns}.\ \mathsf{table\_of}\ \mathit{lvars}\ \mathit{pn} = \mathsf{None}) \wedge\\
\quad\quad (\forall(\mathit{vn},T){\in}\mathsf{set}\ \mathit{lvars}.\ \mathsf{is\_type}\ \Gamma\ T) \wedge\\
\quad\quad (\exists T.\ (\Gamma,\mathsf{table\_of}\ \mathit{lvars}(\mathit{pns}[\mapsto]\mathit{pTs})\ \texttt{(+)}\\
\quad\quad\quad\quad (\mathsf{if\ static}\ m\ \mathsf{then\ empty\ else\ empty}(()\mapsto\mathsf{Class}\ C)))\vdash\\
\quad\quad\quad\quad \mathsf{Body}\ C\ \mathit{blk}\ \mathit{res}{::}{-}T \wedge \Gamma\vdash T\preceq \mathit{rT})
\end{array}
$$

2.9.2 Interfaces

A well-formed interface declaration [GJS96, §9.1] is well-structured (*i.e.* all superinterfaces exist and are not subinterfaces at the same time) and the interface name is not a class name at the same time. All methods actually declared in the interface are uniquely named, well-formed, and non-static. Furthermore, the result type of any method overriding a set of methods defined in the superinterfaces widens to each of the corresponding result types [GJS96, §9.4.1]. So as already observed by Drossopoulou and Eisenbach [DE97a], the result types need not be equal [Generalization $\square \gg \square$]. Note that the specification [GJS96] explicitly allows that an interface inherits more than one method with the same signature. Yet in contrast to the specification, we do not require that these methods have identical return types [Generalization $\square \gg \square$].

$$
\begin{array}{l}
\mathsf{wf\_idecl}\ ::\ \mathit{prog} \to \mathit{idecl} \to \mathit{bool}\\
\mathsf{wf\_idecl}\ \Gamma\ (I,(\mathit{si},\mathit{ms})) \equiv \mathsf{ws\_idecl}\ \Gamma\ I\ \mathit{si} \wedge \neg\mathsf{is\_class}\ \Gamma\ I \wedge\\
\quad\quad \mathsf{unique}\ \mathit{ms} \wedge (\forall(\mathit{sig},\mathit{mh}){\in}\mathsf{set}\ \mathit{ms}.\ \mathsf{wf\_mhead}\ \Gamma\ \mathit{sig}\ \mathit{mh} \wedge \neg\mathsf{static}\ (\mathsf{fst}\ \mathit{mh})) \wedge\\
\quad\quad (\mathsf{o2s} \circ \mathsf{table\_of}\ \mathit{ms}\ \mathsf{hidings}\ \mathsf{Un\_tables}((\lambda J.(\mathsf{imethds}\ \Gamma\ J))\ \text{``}\,\mathsf{set}\ \mathit{si})\\
\quad\quad\ \mathsf{entails}\ (\lambda \mathit{mh}\ (\mathit{md},\mathit{mh}').\ \Gamma\vdash \mathsf{mrt}\ \mathit{mh}\preceq \mathsf{mrt}\ \mathit{mh}'))
\end{array}
$$

2.9.3 Classes

Analogously to interfaces, a well-formed class declaration [GJS96, §8.1] is well-structured and the class name is not an interface name at the same time. All implemented interfaces exist, and for any method of such an interface, the class provides a non-static method implementing it whose return type is in widening (rather than identity, [Generalization $\square \gg \square$]) relation. Like for interfaces, a class may inherit from implemented interfaces more than one method with the same signature, and in contrast to the specification, we do not require that these methods have related return types [Generalization $\square \gg \square$]. All fields and methods actually declared in the class are uniquely named and well-formed. The initialization block is well-typed. Unless the class is `Object`, any method overriding a method of the superclass is static iff the overridden method is static [GJS96, §8.4.6] and has a result type that is in widening (rather than identity, [Generalization $\square \gg \square$]) relation.

$$
\begin{array}{l}
\mathsf{wf\_cdecl}{::}\ \mathit{prog} \to \mathit{cdecl} \to \mathit{bool}\\
\mathsf{wf\_cdecl}\ \Gamma\ (C,(\mathit{sc},\mathit{si},\mathit{fs},\mathit{ms},\mathsf{init})) \equiv \mathsf{ws\_cdecl}\ \Gamma\ C\ \mathit{sc} \wedge \neg\mathsf{is\_iface}\ \Gamma\ C \wedge\\
\quad (\forall I{\in}\mathsf{set}\ \mathit{si}.\ \mathsf{is\_iface}\ \Gamma\ I \wedge (\forall s.\ \forall(\mathit{md}',\mathit{mh}') \in \mathsf{imethds}\ \Gamma\ I\ s.\\
\quad (\exists(\mathit{md}\ ,(\mathit{mh}\ ,b)) \in \mathsf{cmethd}\ \ \Gamma\ C\ s{:}\ \Gamma\vdash \mathsf{mrt}\ \mathit{mh}\preceq \mathsf{mrt}\ \mathit{mh}' \wedge \neg\mathsf{static}\ (\mathsf{fst}\ \mathit{mh})))) \wedge\\
\quad (\forall f{\in}\mathsf{set}\ \mathit{fs}.\ \mathsf{wf\_fdecl}\ \Gamma\ \ f) \wedge \mathsf{unique}\ \mathit{fs} \wedge\\
\quad (\forall m{\in}\mathsf{set}\ \mathit{ms}.\ \mathsf{wf\_mdecl}\ \Gamma\ C\ m) \wedge \mathsf{unique}\ \mathit{ms} \wedge\\
\quad (\Gamma,\mathsf{empty})\vdash \mathsf{init}{::}\surd \wedge\\
\quad (C \neq \texttt{Object} \longrightarrow ((\mathsf{table\_of}\ \mathit{ms})\ \mathsf{hiding}\ (\mathsf{cmethd}\ \Gamma\ \mathit{sc})\ \mathsf{entails}\\
\quad\quad\quad (\lambda(\mathit{mh},b)\ (\mathit{md}',(\mathit{mh}',b')).\ \mathsf{static}\ (\mathsf{fst}\ \mathit{mh}') = \mathsf{static}\ (\mathsf{fst}\ \mathit{mh}) \wedge \Gamma\vdash \mathsf{mrt}\ \mathit{mh}\preceq \mathsf{mrt}\ \mathit{mh}')))
\end{array}
$$

2.9.4 Programs

Finally, all interfaces and classes declared in a well-formed program [GJS96, §8.1, 9.1] are named uniquely and are in turn well-formed. The list of classes includes the class declarations for `Object` and the standard exceptions [Unification].

$$
\begin{array}{l}
\mathsf{wf\_prog} :: \mathit{prog} \rightarrow \mathit{bool} \\
\mathsf{wf\_prog}\ \Gamma \equiv \mathsf{let}\ \mathit{is} = \mathsf{fst}\ \Gamma;\ \mathit{cs} = \mathsf{snd}\ \Gamma\ \mathsf{in} \\
\qquad\qquad \mathsf{ObjectC} \in \mathsf{set}\ \mathit{cs}\ \wedge (\forall \mathit{xn}.\ \mathsf{SXcptC}\ \mathit{xn} \in \mathsf{set}\ \mathit{cs}) \wedge \\
\qquad\qquad (\forall i{\in}\mathsf{set}\ \mathit{is}.\ \ \mathsf{wf\_idecl}\ \Gamma\ i) \wedge \mathsf{unique}\ \mathit{is} \wedge \\
\qquad\qquad (\forall c{\in}\mathsf{set}\ \mathit{cs}.\ \mathsf{wf\_cdecl}\ \Gamma\ c) \wedge \mathsf{unique}\ \mathit{cs}
\end{array}
$$

2.9.5 Properties

There is a whole wealth of properties that can be proved for well-formed programs, most of which are required as lemmas for type soundness. Here we mention just a representative selection.

Many lemmas can be derived straightforwardly from the corresponding definitions, like

$$
\begin{array}{ll}
\mathit{wf\_ws\_prog} & \mathsf{wf\_prog}\ \Gamma \longrightarrow \mathsf{ws\_prog}\ \Gamma \\
\mathit{Object\_is\_class} & \mathsf{wf\_prog}\ \Gamma \longrightarrow \mathsf{is\_class}\ \Gamma\ \mathtt{Object} \\
\mathit{Xcpt\_subcls\_Throwable} & \mathsf{wf\_prog}\ \Gamma \longrightarrow \Gamma \vdash \mathsf{SXcpt}\ \mathit{xn} \preceq_{\mathsf{c}} \mathsf{SXcpt}\ \mathtt{Throwable}
\end{array}
$$

Other lemmas involve induction, typically on the subclass or subinterface hierarchy. For well-formed programs, a class contains (by inheritance) all fields of any superclass:

$$
\begin{array}{ll}
\mathit{fields\_mono} & \begin{array}{l} \mathsf{wf\_prog}\ \Gamma \wedge \mathsf{is\_class}\ \Gamma\ D \wedge \Gamma \vdash D \preceq_{\mathsf{c}} C \wedge \mathsf{table\_of}\ (\mathsf{fields}\ \Gamma\ C)\ \mathit{fn} = \mathsf{Some}\ f \longrightarrow \\ \mathsf{table\_of}\ (\mathsf{fields}\ \Gamma\ D)\ \mathit{fn} = \mathsf{Some}\ f \end{array}
\end{array}
$$

If for some signature a reference type contains a method with that signature, then any class that widens to the reference type (or `Object` if the reference is an array) contains (by inheritance, overriding or hiding) a corresponding method that is static iff the former method is static and has a result type that is in widening relation:

$$
\begin{array}{ll}
 & \mathsf{wf\_prog}\ \Gamma \wedge \mathsf{is\_class}\ \Gamma\ C \wedge \mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ t) \wedge (\mathit{md},\mathit{m},\mathit{pn},\mathit{rT}){\in}\mathsf{mheads}\ \Gamma\ t\ \mathit{sig} \wedge \\
\mathit{class\_mheadsD} & (\mathsf{if}\ (\exists T.\ t = \mathsf{ArrayT}\ \ T)\ \mathsf{then}\ C = \mathtt{Object}\ \mathsf{else}\ \Gamma \vdash \mathsf{Class}\ \ C \preceq \mathsf{RefT}\ t) \longrightarrow \\
 & \exists(\mathit{md}',(\mathit{m}',\mathit{pn}',\mathit{rT}'),\mathit{mb}){\in}\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig}\!:\ \mathsf{static}\ \mathit{m}' = \mathsf{static}\ \mathit{m} \wedge \Gamma \vdash \mathit{rT}' \preceq \mathit{rT}
\end{array}
$$

If an expression is well-typed in the context of a well-formed program, its type is proper:

$$
\mathit{ty\_expr\_is\_type} \quad \mathsf{wf\_prog}\ \Gamma \wedge (\Gamma,\Lambda) \vdash e{::}{-}T \longrightarrow \mathsf{is\_type}\ \Gamma\ T
$$

Chapter 3

Operational Semantics

For formalizing the dynamic aspects of Java$^{\ell ight}$ we have the classical three-fold choice of using an operational, axiomatic, or denotational style.

- An **operational** semantics is close to the Java language specification, rather easy to understand, and more or less directly executable.
- An **axiomatic** semantics aiming at program verification is a bit more abstract, but less intuitive and hard to validate.
- A **denotational** semantics is even more abstract, but also by far more difficult.

Aiming at a simple, easy to validate model (at least for the fundamental description, from which others may be derived), it was a clear decision to prefer the operational style.

The two main parts of the operational semantics are the model of the state and the evaluation rules. The state model will also be used for our axiomatic semantics, as well as most of the auxiliary functions defined along with the evaluation rules.

3.1 State

The program *state* basically consists of the values of all global and local variables. In our model also information on present exceptions is included, whereas more technical auxiliary information like a program counter, a method call stack, and machine registers are not required. See §6.3.3 for example states.

We introduce first a rather generic notion of objects, then global and local stores, exceptions, and finally the overall structure of the program state.

3.1.1 Objects

In Java terminology, an *object* is either a class instance or an array. We model both variants in a uniform way by factoring out the common structure [Factoring $\twoheadrightarrow$], namely a tag and a table of values.

We model the static fields of a class as so-called *class objects* [Reduction $\twoheadrightarrow$]. One could adopt the Smalltalk view that everything is an object, even the set of local variables of method invocations, as done by some language implementations as well. This would reduce redundancy a bit further, but we did not dare to do this in the beginning (at a time when we did not yet handle class objects either), and meanwhile a change of this rather fundamental issue would cost too much, however it is an option worth considering. Such considerations pay since unifying the storage model does not only unify access, which in particular simplifies the notion of conformance (*cf.* §4.3), but also reduces the number of lemmas required for the proof of type soundness.

Object tags are defined via the datatype

```
obj_tag = CInst tname
        | Arr   ty int
```

where for a class instance the class name and for an array its component type and length are given. For a class object the tag will be irrelevant since its type is given already by the reference to it (see below).

An object is a pair of the object tag and a table of values, indexed by variable names *vn*, which are either field specifications (for class instances, as described in §2.8.6) or integers (used as indexes for arrays).

```
vn  = fspec + int
obj = obj_tag × (vn,val)table
```

We define a non-standard selector and a few other access functions on objects:

```
the_Arr :: obj option → ty × int × (vn,val)table
the_Arr obj ≡ ε(T,k,t). obj = Some (Arr T k,t)

upd_obj :: vn → val → obj → obj
upd_obj n v ≡ λ(oi,vs). (oi,vs(n↦v))

obj_ty    :: obj → ty
obj_class :: obj → tname

obj_ty    obj ≡ case fst obj of CInst C → Class C | Arr T k → T[]
obj_class obj ≡ case fst obj of CInst C →       C | Arr T k → Object
```

the_Arr returns the constituents of an optional object that is assumed to be some array, upd_obj updates a variable within an object (*i.e.* a field or array component), obj_ty returns the type of an object, and obj_class returns the class to be used for a method call upon the given object.

We will need object stores for ordinary objects to be referenced via locations on the heap as $(loc,obj)table$, and class objects containing static fields as $(tname,obj)table$. The stores could be defined either in separation, or, isomorphically, in combination as $(loc + tname,obj)table$. We chose the latter option since this enables a uniform access [Unification ≫]. Thus a (generalized) object reference is either a location or a class name:

```
oref = loc + tname
```

We define the abbreviations Heap ≡ Inl and Stat ≡ Inr for clarity.

The table of field types of an object, as determined by its tag and reference, is calculated with the function

```
var_tys :: prog → obj_tag → oref → (vn,ty)table
var_tys Γ oi r ≡ case r of Heap a → (case oi of
                                CInst C → fields_table Γ C (λn (m,fT). ¬static m) (+) empty
                              | Arr T k → empty (+) arr_comps T k)
                         | Stat C → fields_table Γ C (λ(fn,fd) (m,fT). fd = C ∧ static m) (+) empty
```

where

```
i in_bounds  k ≡ 0 ≤ i ∧ i < k
arr_comps T k ≡ λi. if i in_bounds k then Some T else None
fields_table  :: prog → tname → (fspec → field → bool)  → (fspec,ty)table
fields_table Γ C P ≡ option_map snd ∘ table_of (filter (λ(n,f). P n f) (fields Γ C))
```

$\mathsf{fields\_table}$ filters the fields of a class (depending on the predicate P on the field declarations) and returns the corresponding type table. For an instance of a class C these fields are all non-static fields declared in C or inherited from its superclasses, for an array of size k the components with indexes from `0` to k-`1`, and for a class object of a class C the static fields declared in C. The latter ones do not include the static fields inherited from the superclasses of C because such fields are shared and thus need to be stored only once: in the class object of C.

3.1.2 Stores

The program state contains two stores, namely for the (global) objects and the local variables (including method parameters and the This pointer):

$$\begin{aligned} &globs\ = (oref\ \ ,\ obj)table \\ &locals = (lname,\ val)table \end{aligned}$$

We combine them into a type

$$st = \mathsf{st}\ globs\ locals$$

that we use as an abstract datatype in order to make the rest of our model independent of possible future extensions of the state representation. Here an extensive record type would have been helpful.

We denote variables of type st by s, possibly with subscripts.

The (only) operations directly manipulating the stores are

$$\begin{aligned} &\mathsf{globs}\ \ :: st \to globs \\ &\mathsf{locals} :: st \to locals \end{aligned}$$

$$\begin{aligned} &\mathsf{globs}\ \ \equiv \mathsf{st\_rep}\ (\lambda g\ l.\ g) \\ &\mathsf{locals}\ \equiv \mathsf{st\_rep}\ (\lambda g\ l.\ l) \end{aligned}$$

where

$$\mathsf{st\_rep}\ f\ s \equiv \mathsf{case}\ s\ \mathsf{of}\ \mathsf{st}\ g\ l \to f\ g\ l$$

for read access and

$$\begin{aligned} &\mathsf{gupd}(\_\mapsto\_) &&:: &&oref \to obj \to st \to st \\ &\mathsf{lupd}(\_\mapsto\_) &&:: &&lname \to val \to st \to st \\ &\mathsf{upd\_gobj} &&:: oref \to &&vn \to val \to st \to st \\ &\mathsf{set\_locals} &&:: &&locals \to st \to st \end{aligned}$$

$$\begin{aligned} &\mathsf{gupd}(r\mapsto obj) &&\equiv \mathsf{st\_rep}\ (\lambda g\ l.\ \mathsf{st}\ (g(r\mapsto obj))\ l) \\ &\mathsf{lupd}(vn\mapsto v) &&\equiv \mathsf{st\_rep}\ (\lambda g\ l.\ \mathsf{st}\ g\ (l(vn\mapsto v))) \\ &\mathsf{upd\_gobj}\ r\ n\ v &&\equiv \mathsf{st\_rep}\ (\lambda g\ l.\ \mathsf{st}\ (\mathsf{chg\_map}\ (\mathsf{upd\_obj}\ n\ v)\ r\ g)\ l) \\ &\mathsf{set\_locals}\ l &&\equiv \mathsf{st\_rep}\ (\lambda g\ l'.\ \mathsf{st}\ g\ l) \end{aligned}$$

where

$$\mathsf{chg\_map}\ f\ a\ m \equiv \mathsf{case}\ m\ a\ \mathsf{of}\ \mathsf{None} \to m \mid \mathsf{Some}\ b \to m(a\mapsto f\ b)$$

for update and set access.

There are a few derived functions for accessing the $heap = (loc,\ obj)table$ and the This pointer:

$$\begin{aligned} &\mathsf{heap} :: st \to heap \\ &\mathsf{heap}\ s \equiv \mathsf{globs}\ s \circ \mathsf{Heap} \end{aligned}$$

$$\begin{aligned} &\mathsf{lookup\_obj} :: st \to val \to obj \\ &\mathsf{lookup\_obj}\ s\ a' \equiv \mathsf{the}\ (\mathsf{heap}\ s\ (\mathsf{the\_Addr}\ a')) \end{aligned}$$

$\mathsf{val\_this} :: st \to val$
$\mathsf{val\_this}\ s \equiv \mathsf{the}\ (\mathsf{locals}\ s\ \mathsf{This})$

For object initialization we define the functions

$\mathsf{init\_obj} :: prog \to obj\_tag \to oref \to st \to st$
$\mathsf{init\_class\_obj} :: prog \to tname \to st \to st$

$\mathsf{init\_obj}\ \Gamma\ oi\ r \equiv \mathsf{gupd}(r {\mapsto} (oi, \mathsf{init\_vals}\ (\mathsf{var\_tys}\ \Gamma\ oi\ r)))$
$\mathsf{init\_class\_obj}\ \Gamma\ C \equiv \mathsf{init\_obj}\ \Gamma\ \mathsf{arbitrary}\ (\mathsf{Inr}\ C)$

where

$\mathsf{init\_vals} :: (\alpha,\ ty)table \to (\alpha,\ val)table$
$\mathsf{init\_vals}\ vs \equiv \mathsf{option\_map}\ \mathsf{default\_val} \circ vs$

Note that for a class object the object tag is irrelevant (as the corresponding class is already determined by the object reference) and therefore set to arbitrary.

3.1.3 Exceptions

As the word suggests, *exceptions* are exceptional states of a program giving rise to some non-normal mode of execution. We model exceptions naturally as part of the program state, next to the stores, which remain as they are whether an exception is present or not.

In the literature exceptions are sometimes given a more exceptional state than they deserve. For instance, in the transition semantics of Drossopoulou and Eisenbach [DE99] exceptions are regarded as a special form of terms. Thus, a syntactic trick called "expression contexts" has to be used to describe exception propagation in a uniform way. Huisman and Jacobs [HJ00] model the result state of expressions and statements with an outer distinction between hangup, normal completion, and abnormal completion, while giving the store as a parameter where appropriate. This violates the principle of uniformity and thus adds clutter through the omnipresence of case distinctions on the state in their model. Moreover, the axiomatic semantics based on this model uses special kinds of Hoare triples (each with its own version of validity) for reasoning about exceptions, which at least doubles the number of rule variants needed. The different versions of validity have recently been unified [JP00], but the redundancy within each rule remains.

In Java exceptions are represented not as simple values, but as instances of (a subclass of) `Throwable`. This implies that when a standard exception like `NullPointer` is thrown, a suitable object is implicitly allocated[1]. Therefore, apparently spontaneous side-effects on the heap (and — even worse — out-of-memory conditions) may occur almost everywhere, which is a feature rather unpleasant to model. We relieve the pain by representing a thrown standard exception at first by a suitable tag, which is transformed into the corresponding exception object as late as possible: it may become visible to the programmer when reaching a `catch` clause. Consequently, we define the type *xcpt* of exceptions as

$xcpt = \mathsf{XcptLoc}\ loc$
$\quad\mid \mathsf{StdXcpt}\ xname$

where the first alternative references an allocated exception object and the second one represents the intermediate form of a standard exception.

Whether an exception is present — *i.e.* it has been thrown, but not (yet) been caught — is modeled with an optional type:

$xopt = xcpt\ option$

[1]This was not explicitly specified, but appears to be the usual implementation [Clarification □≫□].

In our model many situations arise where under a certain condition an exception should be raised, yet only if no exception is already present which has to take precedence. This behavior is captured by the function

$\mathsf{xcpt\_if} \; :: \; bool \to xopt \to xopt \to xopt$
$\mathsf{xcpt\_if} \; c \; x' \; x \equiv \mathsf{if} \; c \wedge (x = \mathsf{None}) \; \mathsf{then} \; x' \; \mathsf{else} \; x$

It has several typical applications, which we define via the abbreviations

$\mathsf{raise\_if} \; c \; xn \equiv \mathsf{xcpt\_if} \; c \; (\mathsf{Some} \; (\mathsf{StdXcpt} \; xn))$
$\mathsf{np} \; v \qquad\quad \equiv \mathsf{raise\_if} \; (v = \mathsf{Null}) \; \mathtt{NullPointer}$

For example, np v propagates any present exception and otherwise throws the `NullPointer` exception if the value v, which is assumed to be a reference, is the Null pointer.

3.1.4 Full State

The full *program state* is a pair of the exception status and the (global and local) stores:

$state = xopt \times st$

Here a record type would not be very useful because the two components are used and manipulated rather orthogonally. We denote variables of type *state* by σ, possibly with subscripts.

For normal (exception-free) states we provide the abbreviation

$\mathsf{Norm} \; s \equiv (\mathsf{None}, s)$

Conversely, the predicate

$\mathsf{normal} \; \sigma \equiv \mathsf{fst} \; \sigma = \mathsf{None}$

checks if a given state is normal. The predicates

$\mathsf{inited} :: tname \to globs \to bool$
$\mathsf{initd} \;\; :: tname \to state \to bool$

$\mathsf{inited} \; C \; g \equiv g \; (\mathsf{Stat} \; C) \neq \mathsf{None}$
$\mathsf{initd} \;\; C \; \sigma \equiv \mathsf{inited} \; C \; (\mathsf{globs} \; (\mathsf{snd} \; \sigma))$

check whether a given class already has been initialized (or to be exact, initialization is at least in progress), *i.e.* its class object is available.

We further define functionals mapping an update of the exception or store part of the state to an update of the full state,

$\mathsf{xupd} :: (xopt \to xopt) \to state \to state$
$\mathsf{supd} :: (st \quad\;\; \to st \quad\;) \to state \to state$
$\mathsf{xupd} \; f \equiv \lambda(x,s). \; (f \; x, s)$
$\mathsf{supd} \; f \equiv \lambda(x,s). \; (x, f \; s)$

applied for instance when setting and restoring local variables:

$\mathsf{set\_lvars} \qquad :: locals \to state \to state$
$\mathsf{restore\_lvars} :: state \;\; \to state \to state$

$\mathsf{set\_lvars} \; l \qquad\quad\;\; \equiv \mathsf{supd} \; (\mathsf{set\_locals} \; l)$
$\mathsf{restore\_lvars} \; \sigma' \; \sigma \equiv \mathsf{set\_lvars} \; (\mathsf{locals} \; (\mathsf{snd} \; \sigma')) \; \sigma$

3.2 Evaluation

In this section we describe the heart of our semantics: the evaluation rules for Java^{light} terms. We speak of "evaluation" for uniformity (even if, strictly speaking, statements are executed and not evaluated) and for stressing the contrast to transition rules, as discussed next. Then we introduce the general format of our evaluation judgments and introduce our model for abrupt completion (*i.e.* exceptions). Subsequently we give the rules for statements (simple ones, exception handling, class initialization), expressions (simple ones, memory allocation, method call), variables and expression lists. Finally we mention important properties proved for the evaluation relation.

3.2.1 Evaluation *vs.* Transition

When defining an operational semantics, one may choose one (or some combination) of two options: giving

- an **evaluation** (a.k.a. "big-step" or "natural") semantics [Kah87], or
- a **transition** (a.k.a. "small-step" or "structural operational") semantics [Plo81].

We chose an evaluation semantics because of its advantages over a transition semantics (at least for our application):

- It is easier to read and maintain because it is more abstract and less verbose. In particular, it is sufficient to have one rule for each kind of term, rather than a collection of (simpler) rules depending on the number of subterms. This also avoids duplication of side conditions (in positive or negative form) into several of the rules for a single kind of term, and artificial extra side conditions like "ground" and "almost ground" terms [DE99] can be avoided. The low-level nature of transition semantics is particularly striking when given as an Abstract State Machine (ASM) [BS99].
- It is easier to validate since the Java language specification is given in an evaluation-oriented operational style.
- Within complex recursive rules intermediate values need not be stored explicitly, *e.g.* for method calls the current invocation frame does not have to be stored explicitly on a stack (or within the term representing the method body).
- Proofs are much easier to conduct as the powerful principle of rule induction can be used and potentially problematic invariants on intermediate states within execution of one term are not required. Experience with proving type soundness for a transition semantics (*cf.* §4.7.3) particularly confirms this.

On the other hand, the drawbacks of our choice are:

- Multi-threading, *i.e.* concurrency, cannot be described, which is typically done with fine-grained interleaving of transitions.
- Stating a property of infinite executions is not directly possible. Doing so requires the meta-level argument that the property holds for all finite prefixes of them, where the prefixing may be implemented with a counter decremented for each step of evaluation and throwing an exception when zero is reached.

Switching to a transition semantics later would be a drastical change, yet one could re-use the static model and all auxiliary functions, extend the state model as required, and derive the equivalence (for single-threaded programs) of both variants easily.

3.2.2 Judgments

Similarly to the typing judgments, we combine the judgments for the execution of statements and the evaluation of expressions, expression lists, and variables into one. Again we consider statements as a special form of expressions, assigning to them the dummy result value Unit. Here variables cannot be viewed as expressions any more since their result is more complex: it consists of both a simple value (for read access) and a state-transforming function which depends on the value to be assigned to the variable:

$$vvar = val \times (val \to state \to state)$$

The *vvar* notion is reminiscent of *L-values* introduced by Strachey [Str00].

Thus the generalized result type for terms becomes

$$vals = val + vvar + (val)list$$

The general evaluation judgment has the form

$$prog \vdash state \xrightarrow{term \succ} (vals \times state)$$

where $\Gamma \vdash \sigma \xrightarrow{t \succ} (w,\sigma')$ means that in the context of program Γ evaluation of term t from the initial state σ terminates in state σ' and yields the result w.

Analogously to the variants given for the typing judgments, we define the syntactic variants

$$prog \vdash state \xrightarrow{stmt} state$$
$$prog \vdash state \xrightarrow{expr \mathrel{-}\!\succ val} state$$
$$prog \vdash state \xrightarrow{var \mathrel{=}\!\succ vvar} state$$
$$prog \vdash state \xrightarrow{(expr)list \mathrel{\dot{=}}\!\succ (val)list} state$$

by the abbreviations

$$\bullet \equiv \mathsf{In1\ Unit}$$
$$\Gamma \vdash \sigma \xrightarrow{c} \sigma' \;\equiv\; \Gamma \vdash \sigma \xrightarrow{\mathsf{In1r}\ c \succ} (\bullet\ , \sigma')$$
$$\Gamma \vdash \sigma \xrightarrow{e \mathrel{-}\!\succ v} \sigma' \;\equiv\; \Gamma \vdash \sigma \xrightarrow{\mathsf{In1l}\ e \succ} (\mathsf{In1}\ v\ , \sigma')$$
$$\Gamma \vdash \sigma \xrightarrow{e \mathrel{=}\!\succ vf} \sigma' \;\equiv\; \Gamma \vdash \sigma \xrightarrow{\mathsf{In2}\ e \succ} (\mathsf{In2}\ vf, \sigma')$$
$$\Gamma \vdash \sigma \xrightarrow{e \mathrel{\dot{=}}\!\succ v} \sigma' \;\equiv\; \Gamma \vdash \sigma \xrightarrow{\mathsf{In3}\ e \succ} (\mathsf{In3}\ v\ , \sigma')$$

To meet the design goal of minimal redundancy, for each kind of term, we aim to give only one rule capturing all of its behavior. This is possible only if we manage to handle all possible situations in a uniform way. Issues possibly causing rules to be split and techniques to avoid this include:

- explicit alternatives of control flow in conditional statements, which can be dealt with by suitable meta-level conditional expressions
- exceptions possibly present in the start state and intermediate states, for which our approach is described in detail in the next subsection
- (implicit) potential class initializations and similar cases of optional action, which we handle by conditionally executing either the appropriate statement(s) or Skip[2].

Using the techniques just mentioned, we succeed to avoid splitting rules.

[2]Because of limitations of the inductive definition package of HOL, we cannot replace a judgment like $\Gamma \vdash \sigma \xrightarrow{\mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else\ Skip}} \sigma'$ by the equivalent — and perhaps a bit more readable — if b then $\Gamma \vdash \sigma \xrightarrow{c} \sigma'$ else $\sigma = \sigma'$ since in this term the inductively defined relation appears conditionally.

A related subtle and conceptually problematic issue is how to deal with expectations on the dynamic types of intermediate values. Take as a simple example the statement `if`(e) c_1 `else` c_2. Its execution relies on the assumption that the value v of e is a Boolean value and not, say, an object reference. This brings us already to the concept of type soundness, which we can handle only after we have defined the operational semantics itself. There are at least three ways to solve this problem:

"aggressive": Simply assume that everything is in order and use selectors like $\mathsf{the\_Bool}$ that yield an unknown result if they happen to be applied to *e.g.* $\mathsf{Intg}\ 42$. For example, for the conditional statement we can give the rule

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1 \quad \Gamma\vdash \sigma_1 \xrightarrow{(\mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2} \sigma_2}$$

that is applicable even if v is not a Boolean value, in which case either c_1 or c_2 is selected arbitrarily. This solution not only leads to simple proofs but also has the special conceptual advantage of being closest to the actual behavior of execution.

"strict": If something is wrong, let evaluation get stuck, which is modeled by the situation that no rule is applicable. In our example, the rule

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ \mathsf{Bool}\ b} \sigma_1 \quad \Gamma\vdash \sigma_1 \xrightarrow{(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2} \sigma_2}$$

can be applied (with pattern matching) only if the outcome of e is indeed a Boolean value, and there is no further rule for the conditional statement. For an evaluation semantics this implies that getting stuck and non-termination cannot be distinguished, and thus type soundness cannot be formulated at all. Moreover, even for a transition semantics, type soundness becomes a progress property, *i.e.* the existence of a final (next) state has to be proved. Hence the transition relation occurs positively in the soundness theorem and the convenient rule induction scheme cannot be used.

"defensive": In case of errors throw some (non-catchable) "TypeMismatch" exception, such that the proof of type soundness has to show that this exception will never be actually thrown. In our example we could either take the rule just given and add the extra rule

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2} \mathsf{xupd}\ (\mathsf{raise\_if}\ \mathsf{True}\ \mathtt{TypeMismatch})\ \sigma_1}$$

or give the combined rule

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1 \qquad \Gamma\vdash \mathsf{xupd}\ (\mathsf{raise\_if}\ (\neg\exists b.\ v{=}\mathsf{Bool}\ b)\ \mathtt{TypeMismatch})\ \sigma_1 \xrightarrow{(\mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2} \sigma_2}$$

The disadvantage is that this approach leads to new rule variants or at least complicates the existing rules through additional case distinctions.

From the argumentation given it will be obvious that we opted for the first variant.

3.2.3 Exception Propagation

When an exception is thrown, any subsequent computation is skipped and the exception is propagated until it is caught or the program execution terminates.

For an evaluation semantics the standard way of implementing propagation is as follows. Exceptions are assumed to be present only in the final state of judgments, but never in the initial state. Thus the judgments have the general form $st \xrightarrow{term\succ} state$ (abstracting from irrelevant details here). All intermediate states within an evaluation rule potentially contain exceptions. Therefore the rules have to be split at all these intermediate states to distinguish the two possible situations where an exception is present (which has to be propagated) or not. For sequential composition of two subterms, which is the simplest example, this looks like

$$\frac{s_0 \xrightarrow{t_1\succ} (\mathsf{None}, s_1) \qquad s_1 \xrightarrow{t_2\succ} \sigma_2}{\sigma_0 \xrightarrow{t_1;\ t_2\succ} \sigma_2}$$

$$\frac{s_0 \xrightarrow{t_1\succ} (\mathsf{Some}\ xs, s_1)}{s_0 \xrightarrow{t_1;\ t_2\succ} (\mathsf{Some}\ xs, s_1)}$$

Consequently rules for a term with n subterms have to be split into at least n rules since it involves at least $n-1$ intermediate states. In Java there are many terms with more than one subterm, so we were highly motivated to invent a better solution.

Our solution reminds of monads in the sense that exceptions are propagated behind the scenes: the above explicit case distinction on the caller's side of a rule is moved to the callee's side and made more or less implicit. To this end, exceptions are permitted also in the start state of judgments (incidentally yielding more symmetry). Now, there is one general rule defining exception propagation:

$$\frac{}{(\mathsf{Some}\ xs, s) \xrightarrow{t\succ} (\mathsf{Some}\ xs, s)}$$

All further evaluation rules can assume that in their initial states no exception is present. That is, they have the generic form

$$\frac{\mathsf{Norm}\ s_0 \xrightarrow{t_1\succ} \sigma_1 \qquad \sigma_1 \xrightarrow{t_2\succ} \ldots}{\mathsf{Norm}\ s_0 \xrightarrow{t\succ} \ldots}$$

In our specific model, the full rule for exception propagation is

$$14.1, 15.5^3 \quad \frac{}{\Gamma \vdash (\mathsf{Some}\ xc, s) \xrightarrow{t\succ} (\mathsf{arbitrary3}\ t, (\mathsf{Some}\ xc, s))}$$

What we have added here is production of a suitable result value. In case an exception is present, the result value entry in judgments is irrelevant, but its full inclusion helps to make the structure independent of exception occurrence [Unification ≫]. Such irrelevant values are normally ignored, so it should not matter whether they are unique. Yet for simplicity we prefer (fixed) arbitrary values over "nondeterministic" values. Additionally, our unified model for expressions, variables *etc.* requires that they are at least of the corresponding type, *i.e.* involve the correct injection. This is achieved by the auxiliary function

```
arbitrary3 :: term → vals
arbitrary3 t ≡ case t of In1 ec → In1 (case ec of Inl e → Inl arbitrary | Inr c → Inr Unit)
                       | In2 v  → In2 arbitrary
                       | In3 es → In3 arbitrary
```

[3] The numeric labels appearing in this and many subsequent rules refer to the corresponding definition in [GJS96].

3.2.4 Standard Statements

Due to our implicit exception propagation mechanism, the rules for those statements not directly involving exceptions appear almost as usual:

$$14.5\ \frac{}{\Gamma\vdash \mathsf{Norm}\ s \xrightarrow{\mathsf{Skip}} \mathsf{Norm}\ s} \qquad 14.2\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{c_1} \sigma_1 \quad \Gamma\vdash \sigma_1 \xrightarrow{c_2} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{c_1;\ c_2} \sigma_2}$$

$$14.7\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Expr}\ e} \sigma_1} \qquad 14.8.2\ \frac{\begin{array}{c}\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ b} \sigma_1 \\ \Gamma\vdash \sigma_1 \xrightarrow{(\mathsf{if\ the\_Bool}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)} \sigma_2\end{array}}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2} \sigma_2}$$

$$14.10\ \frac{\begin{array}{c}\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ b} \sigma_1 \\ \mathsf{if\ the\_Bool}\ b\ \mathsf{then}\ \Gamma\vdash \sigma_1 \xrightarrow{c} \sigma_2 \wedge \Gamma\vdash \sigma_2 \xrightarrow{\mathtt{while}(e)\ c} \sigma_3\ \mathsf{else}\ \sigma_3{=}\sigma_1\end{array}}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{while}(e)\ c} \sigma_3}$$

The three conditions of the loop rule could have been combined yielding the more compact rule

$$14.10\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{if}(e)\ (c;\ \mathtt{while}(e)\ c)\ \mathtt{else}\ \mathsf{Skip}} \sigma_3}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{while}(e)\ c} \sigma_3}$$

We decided not to do so for uniformity with the other rules (such that all judgments in the preconditions of any rule are non-composite Java$^{\ell ight}$ terms), which is also helpful for proofs. Of course, one version of the rule can be derived from the other.

3.2.5 Exception Handling

Throwing a user-defined exception means first evaluating the reference.

$$14.16\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ a'} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{throw}\ e} \mathsf{xupd}\ (\mathsf{throw}\ a')\ \sigma_1}$$

If no exception has occurred doing this and the resulting value is not a null reference[4], the auxiliary function throw copies the evaluated location into the exception component of the state:

$\mathsf{throw} :: val \to xopt \to xopt$
$\mathsf{throw}\ a'\ x \equiv \mathsf{xcpt\_if\ True\ (Some\ (XcptLoc\ (the\_Addr}\ a')))\ (\mathsf{np}\ a'\ x)$

When describing the effect of the statement $\mathtt{try}\ c_1\ \mathtt{catch}(C\ vn)\ c_2$ we have to distinguish whether in state σ_2 after execution of c_1 an exception of appropriate (dynamic) type, *viz.* a subclass of C, is present, as denoted by

$\_,\_\vdash \mathsf{catch}\ \_ :: prog \to state \to tname \to bool$
$\Gamma,\sigma\vdash \mathsf{catch}\ C \equiv \exists xc.\ \mathsf{fst}\ \sigma = \mathsf{Some}\ xc \wedge \Gamma,\mathsf{snd}\ \sigma\vdash \mathsf{Addr\ (the\_XcptLoc}\ xc)\ \mathsf{fits\ Class}\ C$

This predicate in turn relies on

$\_,\_\vdash \_\ \mathsf{fits}\ \_ :: prog \to st \to val \to ty \to bool$
$\Gamma,s\vdash a\ \mathsf{fits}\ T \equiv (\exists rt.\ T = \mathsf{RefT}\ rt) \longrightarrow a = \mathsf{Null} \vee \Gamma\vdash \mathsf{obj\_ty\ (lookup\_obj}\ s\ a)\preceq T$

[4]This test of *null* pointer dereferencing was not mentioned in specification, [Clarification $\Box\!\gg\!\Box$]

checking whether a value a is assignable to type T. The fits predicate will be used also for the dynamic type checks in type casts, the `instanceof` expression, and array assignments. In all these applications, if T is not a reference type, it is known from the context that the assignment is harmless and so the predicate yields True. Thus it is strictly weaker than the notion of conformance introduced in §4.3.

$$14.18.1\quad \frac{\begin{array}{c}\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{c_1} \sigma_1 \qquad \Gamma\vdash \sigma_1 \xrightarrow{\mathsf{sxalloc}} \sigma_2\\ \mathsf{if}\ \Gamma,\sigma_2\vdash \mathsf{catch}\ C\ \mathsf{then}\ \Gamma\vdash \mathsf{new\_xcpt\_var}\ vn\ \sigma_2 \xrightarrow{c_2} \sigma_3\ \mathsf{else}\ \sigma_3{=}\sigma_2\end{array}}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{try}\ c_1\ \mathtt{catch}(C\ vn)\ c_2} \sigma_3}$$

In case $\Gamma,\sigma_2\vdash$catch C holds, the statement c_2 of the `catch` clause is executed with its exception parameter vn set to the caught exception using the function

$\mathsf{new\_xcpt\_var} :: ename \to state \to state$
$\mathsf{new\_xcpt\_var}\ vn \equiv \lambda(x,s).\ \mathsf{Norm}\ (\mathsf{lupd}(\mathsf{EName}\ vn{\mapsto}\mathsf{Addr}\ (\mathsf{the\_XcptLoc}\ (\mathsf{the}\ x)))\ s)$

After the `catch` clause the exception parameter is still present (as a local variable in the state), which is harmless because it becomes inaccessible.

As motivated in §3.1.3, standard exceptions of class xn may be represented by StdXcpt xn, which must be replaced by XcptLoc a before entering the `catch` clause, where a is the location of a newly allocated instance of class xn. This task is performed by the (partial) function sxalloc, which we define inductively as a judgment of the form

$prog\vdash state \xrightarrow{\mathsf{sxalloc}} state$

for analogy with the evaluation judgments:

$$\frac{}{\Gamma\vdash \mathsf{Norm}\ s \xrightarrow{\mathsf{sxalloc}} \mathsf{Norm}\ s} \qquad \frac{}{\Gamma\vdash (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),s) \xrightarrow{\mathsf{sxalloc}} (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),s)}$$

$$\frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{halloc}\ (\mathsf{CInst}\ (\mathsf{SXcpt}\ xn))\succ a} (x,s_1)}{\Gamma\vdash (\mathsf{Some}\ (\mathsf{StdXcpt}\ xn),s_0) \xrightarrow{\mathsf{sxalloc}} (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),s_1)}$$

If no standard exception is present, the function performed is the identity on the state. Otherwise, the judgment halloc allocates the exception object (if possible). See §6.3.3 for an application example, and memory allocation will be described in §3.2.8. Our approach using StdXcpt and sxalloc my seem complicated, but note that a more direct model would be even more complicated: allocating an exception causes a side-effect on the heap, which would add much clutter to the already rather complex (and sometimes even nested, see *e.g.* the definition of avar in §3.2.10) conditional expressions defining the generation of standard exceptions.

The `finally` statement is similar to the sequential composition, but executes its second clause from a normal state regardless whether an exception has been thrown in its first clause or not. If one exception occurs in either clause, it is (re-)raised after the statement, and if both parts throw an exception, the second one takes precedence.

$$14.18.2\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{c_1} (x_1,s_1) \qquad \Gamma\vdash \mathsf{Norm}\ s_1 \xrightarrow{c_2} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{c_1\ \mathtt{finally}\ c_2} \mathsf{xupd}\ (\mathsf{xcpt\_if}\ (x_1{\neq}\mathsf{None})\ x_1)\ \sigma_2}$$

3.2.6 Class Initialization

Within the evaluation of a few expressions, *e.g.* field accesses and method calls, *first active use* of some class C is possible triggering its initialization. To model this behavior we insert an artificial statement init C at those positions. We had to fix the exact positions, but we are not sure if this should be considered as an improvement of the specification [Clarification

$\square\!\!\gg\!\square$] or if the positions have been left unspecified intentionally. If the class in question is already initialized, below abbreviated by *initedC*, there is nothing to do. Otherwise, a new class object is allocated — incidentally marking that initialization is in progress —, and if the class is not `Object`, its superclass is (potentially) initialized. Then the static initializer of the current class is executed, whereby the current local variables have to be hidden and afterwards restored.

$$12.4.2, 8.5\quad \frac{\begin{array}{c}\mathsf{the}\ (\mathsf{class}\ \Gamma\ C) = (sc,si,fs,ms,ini) \quad \mathsf{if\ inited}\ C\ (\mathsf{globs}\ s_0)\ \mathsf{then}\ \sigma_3 = \mathsf{Norm}\ s_0\ \mathsf{else} \\ (\Gamma\vdash \mathsf{Norm}\ (\mathsf{init\_class\_obj}\ \Gamma\ C\ s_0)\ \xrightarrow{(\mathsf{if}\ C = \mathtt{Object}\ \mathsf{then\ Skip\ else\ init}\ sc)} \sigma_1\ \wedge \\ \Gamma\vdash \mathsf{set\_lvars\ empty}\ \sigma_1 \xrightarrow{ini} \sigma_2\ \wedge\ \sigma_3 = \mathsf{restore\_lvars}\ \sigma_1\ \sigma_2)\end{array}}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ C} \sigma_3}$$

We ignore the rare case of memory overflow when allocating class objects (in contrast ordinary objects) [Restriction $\square\!\!\gg\!\square$].

3.2.7 Simple Expressions

In contrast to the statement rules, most evaluation rules for expressions deserve a comment.

The result of a literal expression is simply the given value, and the value of `super` is that of `this`. The state is left unchanged.

$$15.7.1\quad \frac{}{\Gamma\vdash \mathsf{Norm}\ s \xrightarrow{\mathsf{Lit}\ v -\succ v} \mathsf{Norm}\ s} \qquad 15.10.2\quad \frac{}{\Gamma\vdash \mathsf{Norm}\ s \xrightarrow{\mathtt{super} -\succ \mathsf{val\_this}\ s} \mathsf{Norm}\ s}$$

A (reading) variable access returns the value of the variable:

$$15.2\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{va =\succ (v,f)} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Acc}\ va -\succ v} \sigma_1}$$

Variable assignment evaluates the right hand side and uses its value to (possibly) update the state:

$$15.25.1\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{va =\succ (w,f)} \sigma_1 \qquad \Gamma\vdash \sigma_1 \xrightarrow{e -\succ v} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{va \mathtt{:=} e -\succ v} \mathsf{assign}\ f\ v\ \sigma_2}$$

The update takes place only if no exception is already present and the update function itself does not throw a new one, as implemented by

$$\begin{array}{ll}\mathsf{assign} & ::\ (val \to state \to state) \to val \to state \to state \\ \mathsf{assign}\ f\ v \equiv \lambda(x,s).\ \mathsf{let}\ (x',s') = \mathsf{if}\ x = \mathsf{None\ then}\ f\ v\ (x,s)\ \mathsf{else}\ (x,s) \\ \qquad\qquad\qquad\quad \mathsf{in}\ \ (x', \mathsf{if}\ x' = \mathsf{None\ then}\ s'\ \mathsf{else}\ s)\end{array}$$

The semantics of the conditional expression is straightforward:

$$15.24\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e_0 -\succ b} \sigma_1 \qquad \Gamma\vdash \sigma_1 \xrightarrow{(\mathsf{if\ the\_Bool}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) -\succ v} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e_0\ \mathtt{?}\ e_1\ \mathtt{:}\ e_2 -\succ v} \sigma_2}$$

A type cast merely evaluates its argument and raises an exception if the dynamic type of the result happens to be unsuitable:

$$15.15\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e -\succ v} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Cast}\ T\ e -\succ v} \mathsf{xupd}\ (\mathsf{raise\_if}\ (\neg(\Gamma,\mathsf{snd}\ \sigma_1 \vdash v\ \mathsf{fits}\ T))\ \mathtt{ClassCast}\ )\ \sigma_1}$$

Similarly, the type comparison operator flags whether the type of its argument is assignable to the given reference type:

$$15.19.2\quad \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e -\succ v} \sigma_1}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e\ \mathtt{instanceof}\ T -\succ \mathsf{Bool}\ (v \neq \mathsf{Null}\ \wedge\ \Gamma,\mathsf{snd}\ \sigma_1 \vdash v\ \mathsf{fits\ RefT}\ T)} \sigma_1}$$

3.2.8 Memory Allocation

Memory allocation is an intricate issue: its outcome is practically (from the user's perspective) non-deterministic, and it may even fail. As we intend to model this behavior faithfully and as loosely as possible, we use the following allocation function:

$$\begin{array}{l} \mathsf{new\_Addr} \quad :: \mathit{heap} \rightarrow \mathit{loc\ option} \\ \mathsf{new\_Addr}\ h \equiv \mathsf{if}\ (\forall a.\ h\ a \neq \mathsf{None})\ \mathsf{then\ None\ else\ Some}\ (\varepsilon a.\ h\ a = \mathsf{None}) \end{array}$$

If there is no free location on the heap, it fails, otherwise returns a free location. Here Hilbert's choice operator is used naturally. Note that since its result is determined by the heap, but not actually known, new_Addr is in fact just a — non-executable — specification [Underspec $\square \gg \square$]. Of course more detailed — and executable — models could be given, for example instantiating the type of locations to a fixed range of natural numbers where new_Addr returns the lowest such number not occupied (if any). See §6.3.3 for an example application.

Heap Extension

The "partial" function new_Addr is the heart of our allocation judgment

$$\mathit{prog} \vdash \mathit{state} \xrightarrow{\mathsf{halloc}\ \mathit{obj\_tag} \succ \mathit{loc}} \mathit{state}$$

which we write (and define) in the style of the evaluation judgments because of its strong relation to them. If possible, halloc allocates a fresh object with the given tag on the heap, initializes it (using init_obj), and returns an updated state. Otherwise it raises an `OutOfMemory` exception.

$$12.5 \quad \frac{\begin{array}{c}\mathsf{new\_Addr}\ (\mathsf{heap}\ s) = \mathsf{Some}\ a \qquad (x, oi') = (\mathsf{if\ atleast\_free}\ (\mathsf{heap}\ s)\ 2\ \mathsf{then}\ (\mathsf{None}, oi)) \\ \mathsf{else}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ a), \mathsf{CInst}\ (\mathsf{SXcpt}\ \mathtt{OutOfMemory})))\end{array}}{\Gamma \vdash \mathsf{Norm}\ s \xrightarrow{\mathsf{halloc}\ oi \succ a} (x, \mathsf{init\_obj}\ \Gamma\ oi'\ (\mathsf{Heap}\ a)\ s)}$$

If an exception is present already in the start state, it is propagated as usual:

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ x, s) \xrightarrow{\mathsf{halloc}\ oi \succ \mathsf{arbitrary}} (\mathsf{Some}\ x, s)}$$

Interference with Exception Object Creation

A further complication arising from potential memory exhaustion is due to the fact that exception objects themselves have to be allocated. [GJS96] does not specify what happens if there is not enough memory even to allocate an `OutOfMemory` exception. We model a sensible implementation [Clarification $\square \gg \square$] that signals success of memory allocation only if there is still another free location left on the heap. The situation where there is enough memory to allocate not only the object in question, but also a further (possibly exception) object, is formalized as

$\mathsf{atleast\_free}\ (\mathsf{heap}\ s)\ 2$

where[5]

$$\begin{array}{l} \mathsf{atleast\_free} :: \mathit{heap} \rightarrow \mathit{nat} \rightarrow \mathit{bool} \\ \mathsf{atleast\_free}\ h\ \ 0 \qquad = \mathsf{True} \\ \mathsf{atleast\_free}\ h\ (n{+}1) = (\exists a.\ h\ a = \mathsf{None} \wedge (\forall \mathit{obj}.\ \mathsf{atleast\_free}\ (h(a \mapsto \mathit{obj}))\ n)) \end{array}$$

Otherwise (*i.e.* if no further location is left) halloc already returns an `OutOfMemory` exception, such that when the corresponding exception object has to be allocated (using sxalloc in our case), this is still possible.

[5]Interestingly, the actual definition does not matter for our meta-theoretical proofs. Thus, we are free to abstract also from the actual (byte) size of objects [Underspec $\square \gg \square$].

Note that sxalloc (*cf.* §3.2.5) is designed carefully to implement the behavior just described: it calls halloc from a normal state, and if an `OutOfMemory` exception has already been thrown because there is just one free location left, halloc yields that location (and throws an `OutOfMemory` exception again, which is ignored). sxalloc uses this last free location to allocate the required instance of `OutOfMemory`. Yet if that exception is caught later and any further attempt to allocate objects is made [6], we have to give up: we simply stop execution, which we model with the premise $\mathsf{new\_Addr}\ (\mathsf{heap}\ s) = \mathsf{Some}\ a$, rendering the halloc rule non-applicable in that case.

Object Creation

Now that the primitives for memory allocation have been developed, it is straightforward to apply them for class instance and array creation:

$$15.8.1, 12.4.1\ \frac{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ C} \sigma_1 \qquad \Gamma \vdash \sigma_1 \xrightarrow{\mathsf{halloc}\ (\mathsf{CInst}\ C) \succ a} \sigma_2}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{new}\ C - \succ \mathsf{Addr}\ a} \sigma_2}$$

Just note that for the expressions `new` C and `new` $T[e]$ (in case T is a class) the class object of C potentially has to be initialized first. For array creation, in addition the component count is evaluated and a suitable exception is thrown if the result is negative.

$$15.9.1, 12.4.1\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init\_comp\_ty}\ T} \sigma_1 \qquad \Gamma \vdash \sigma_1 \xrightarrow{e - \succ i'} \sigma_2 \\ \Gamma \vdash \mathsf{xupd}\ (\mathsf{check\_neg}\ i')\ \sigma_2 \xrightarrow{\mathsf{halloc}\ (\mathsf{Arr}\ T\ (\mathsf{the\_Intg}\ i')) \succ a} \sigma_3\end{array}}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{new}\ T[e] - \succ \mathsf{Addr}\ a} \sigma_3}$$

where

$\mathsf{init\_comp\_ty} :: ty \to stmt$
$\mathsf{init\_comp\_ty}\ T \equiv \mathsf{if}\ (\exists C.\ T = \mathsf{Class}\ C)\ \mathsf{then\ init}\ (\mathsf{the\_Class}\ T)\ \mathsf{else\ Skip}$
$\mathsf{check\_neg}\ i' \quad \equiv \mathsf{raise\_if}\ (\mathsf{the\_Intg}\ i'\ ``````{<}0)\ \mathtt{NegArrSize}$

We do not consider garbage collection [Restriction $\square \gg \square$], therefore there is no need to model finalizers.

3.2.9 Method Call

A method call evaluates the target reference and the arguments, determines the target class, sets the local variables of the callee, transfers control, and finally restores the local variables:

$$15.11.4\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{e - \succ a'} \sigma_1 \qquad \Gamma \vdash \sigma_1 \xrightarrow{args \doteq \succ vs} \sigma_2 \qquad C = \mathsf{target}\ mode\ (\mathsf{snd}\ \sigma_2)\ a'\ md \\ \Gamma \vdash \mathsf{init\_lvars}\ \Gamma\ C\ (mn,pTs)\ mode\ a'\ vs\ \sigma_2 \xrightarrow{\mathsf{Methd}\ C\ (mn,pTs) - \succ v} \sigma_3\end{array}}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\{t,md,mode\}e..mn(\{pTs\}args) - \succ v} (\mathsf{restore\_lvars}\ \sigma_2\ \sigma_3)}$$

Note that we do not model potential stack overflow [Restriction $\square \gg \square$]. See §6.3.3 for an application example. The function

$\mathsf{target} :: inv\_mode \to st \to val \to ref\_ty \to tname$
$\mathsf{target}\ m\ s\ a'\ rt \equiv \mathsf{if}\ m = \mathsf{IntVir}\ \mathsf{then\ obj\_class}\ (\mathsf{lookup\_obj}\ s\ a')\ \mathsf{else\ the\_Class}\ (\mathsf{RefT}\ rt)$

computes the target according to the invocation mode. For invocation mode `interface` or `virtual`, it dynamically looks up the class of the object, and for mode `static` or `super`, it returns the type *md* in the annotation (statically) determined by the well-typedness rule

[6] According to our experiments, in this strange situation current Java implementations show a wide range of behavior: from sudden termination without executing `finally` blocks, over hangup, to infinite invocation of a single exception handler.

(*cf.* §2.8.5). Since invocation mode `super` uses the superclass of the caller's class, it is effectively static [Correction $\gg$]. Still we have to distinguish `super` from `static` because only for the latter case the This pointer is not set.

The auxiliary function init_lvars primarily assigns the argument values to the parameter variables. If the invocation mode is not `static`, it also assigns target reference of the call to the This pointer and checks if the reference is the Null pointer. Furthermore, it initializes the (remaining) local variables with their default values. This is necessary for type safety as we do not model the *definite assignment* check [GJS96, §16] [Restriction $\gg$]. Yet if a program passes the check (and assuming that the check is designed and implemented correctly) then it will read local variables only after explicitly assigning to them. The artificial initial values that we introduce are thus unobservable.

$$
\begin{array}{l}
\mathsf{init\_lvars} :: \mathit{prog} \rightarrow \mathit{tname} \rightarrow \mathit{sig} \rightarrow \mathit{inv\_mode} \rightarrow \mathit{val} \rightarrow \mathit{val\ list} \rightarrow \mathit{state} \rightarrow \mathit{state}\\
\mathsf{init\_lvars}\ \Gamma\ C\ \mathit{sig}\ \mathit{mode}\ a'\ \mathit{pvs} \equiv \lambda(x,s).\ \mathsf{let}\\
\qquad (\_,(\_,\mathit{pns},\_),\mathit{lvars},\_) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig});\\
\qquad l = \mathsf{init\_vals}(\mathsf{table\_of}\ \mathit{lvars})(\mathit{pns}[\mapsto]\mathit{pvs})\ \texttt{(+)}\\
\qquad\quad (\mathsf{if}\ \mathit{mode} = \mathsf{Static}\ \mathsf{then}\ \mathsf{empty}\ \mathsf{else}\ \mathsf{empty}(()\mapsto a'))\\
\qquad \mathsf{in}\ \mathsf{set\_lvars}\ l\ (\mathsf{if}\ \mathit{mode} = \mathsf{Static}\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{np}\ a'\ x,s)
\end{array}
$$

Note that the method lookup as modeled by $\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig}$ does not need to take the return type into account, other than stated in [GJS96, §15.11.4.4] [Correction $\gg$].

As will be motivated in §5.6.8, we distinguish the callee's side of method calls, the method implementation, from the caller's side and further handle the actual method body separately. This also helps to keep the complexity of the method call rule bearable. Thus the only thing to do for the method implementation rule is to look up the method according to its signature and determine the information needed for the body:

$$
\frac{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{body}\ \Gamma\ C\ \mathit{sig} \multimap\!\succ v} \sigma_1}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Methd}\ \ C\ \mathit{sig} \multimap\!\succ v} \sigma_1}
$$

where $\mathsf{body}\ \Gamma\ C\ \mathit{sig} = \mathsf{let}\ (D,\_,\_,c,e) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ C\ \mathit{sig})\ \mathsf{in}\ \mathsf{Body}\ D\ c\ e$

The evaluation of the body is just a sequential composition of initializing the current class (if required), executing the block of statements, and evaluating the result expression:

$$
14.15, 12.4.1\ \frac{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ D} \sigma_1 \qquad \Gamma \vdash \sigma_1 \xrightarrow{c} \sigma_2 \qquad \Gamma \vdash \sigma_2 \xrightarrow{e \multimap\!\succ v} \sigma_3}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Body}\ D\ c\ e \multimap\!\succ v} \sigma_3}
$$

One could alternatively reduce Body to the sequential composition of its three components using ; [Reduction $\gg$], which would require a slight generalization of ; to enable propagation of the result v.

3.2.10 Variables

As already motivated in §2.4.3, and specified in §3.2.2, the result of a variable consists of its current value and an update function. Since the calculation of the result is rather involved and identical for the operational semantics and axiomatic semantics, we define it using auxiliary functions, one for each kind of variable.

The simplest case is of course the one for local variables:

$$
15.13.1, 15.7.2\ \frac{}{\Gamma \vdash \mathsf{Norm}\ s \xrightarrow{\mathsf{LVar}\ \mathit{vn} =\!\succ \mathsf{lvar}\ \mathit{vn}\ s} \mathsf{Norm}\ s}
$$

where

$$
\begin{array}{l}
\mathsf{lvar} :: \mathit{lname} \rightarrow \mathit{st} \rightarrow \mathit{vvar}\\
\mathsf{lvar}\ \mathit{vn}\ s \equiv (\mathsf{the}\ (\mathsf{locals}\ s\ \mathit{vn}),\ \lambda v.\ \mathsf{supd}\ (\mathsf{lupd}(\mathit{vn} \mapsto v)))
\end{array}
$$

The rule for field variables is more involved, for several reasons: It has to consider class initialization, check for Null pointer access, and access the object store distinguishing static and instance variables. At least we avoid much redundancy by sharing most parts of the computation necessary for read and write access.

$$15.10.1, 12.4.1\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ C} \sigma_1 \quad \Gamma\vdash\sigma_1 \xrightarrow{e-\succ a'} \sigma_2 \quad (v,\sigma_2') = \mathsf{fvar}\ C\ stat\ fn\ a'\ \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{\{C,stat\}e..fn=\succ v} \sigma_2'}$$

where

$\mathsf{fvar} :: tname \to bool \to ename \to val \to state \to vvar \times state$
$\mathsf{fvar}\ C\ stat\ fn\ a'\ \sigma \equiv \mathsf{let}\ (oref,xf) = \mathsf{if}\ stat\ \mathsf{then}\ (\mathsf{Stat}\ C,\mathsf{id})\ \mathsf{else}\ (\mathsf{Heap}\ (\mathsf{the\_Addr}\ a'),\mathsf{np}\ a');$
$n = \mathsf{Inl}\ (fn,C);\ f = (\lambda v.\ \mathsf{supd}\ (\mathsf{upd\_gobj}\ oref\ n\ v))$
$\mathsf{in}\ ((\mathsf{the}\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{globs}\ (\mathsf{snd}\ \sigma)\ oref))\ n),f),\mathsf{xupd}\ xf\ \sigma)$

Even more complex are array variables because of several cases where an exception has to be thrown: a check for Null pointer dereferencing and index bound violation is performed, which for simplicity is done before evaluating the right-hand side. Furthermore, the update function activated for write access by the assignment rule performs a dynamic type check on the value to be stored.

$$15.12.1, 15.25.1\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e_1-\succ a} \sigma_1 \quad \Gamma\vdash\sigma_1 \xrightarrow{e_2-\succ i} \sigma_2 \quad (v,\sigma_2') = \mathsf{avar}\ \Gamma\ i\ a\ \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e_1[e_2]=\succ v} \sigma_2'}$$

where

$\mathsf{avar} :: prog \to val \to val \to state \to vvar \times state$
$\mathsf{avar}\ \Gamma\ i'\ a'\ \sigma \equiv \mathsf{let}\ oref = \mathsf{Heap}\ (\mathsf{the\_Addr}\ a');\ \ i = \mathsf{the\_Intg}\ i';$
$n = \mathsf{Inr}\ i;\ \ (T,k,cs) = \mathsf{the\_Arr}\ (\mathsf{globs}\ (\mathsf{snd}\ \sigma)\ oref);$
$f = (\lambda v\ (x,s).\ (\mathsf{raise\_if}\ (\neg\Gamma,s\vdash v\ \mathsf{fits}\ T)\ \mathtt{ArrStore}\ x,\ \mathsf{upd\_gobj}\ oref\ n\ v\ s))$
$\mathsf{in}\ ((\mathsf{the}\ (cs\ n),f),\ \mathsf{xupd}\ (\mathsf{raise\_if}\ (\neg\ i\ \mathsf{in\_bounds}\ k)\ \mathtt{IndOutBound} \circ \mathsf{np}\ a')\ \sigma)$

3.2.11 Expression Lists

The evaluation of expressions canonically extends to expression lists:

$$\frac{}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{[]\doteq\succ[]} \mathsf{Norm}\ s_0}$$

$$15.6.4\ \frac{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1 \quad \Gamma\vdash\sigma_1 \xrightarrow{es\doteq\succ vs} \sigma_2}{\Gamma\vdash \mathsf{Norm}\ s_0 \xrightarrow{e\#es\doteq\succ v\#vs} \sigma_2}$$

3.2.12 Properties

In analogy to the typing judgments, the injections used in the evaluation rules do not mismatch:

$$\begin{array}{l} \Gamma\vdash s \xrightarrow{t\succ} (w,s') \longrightarrow \mathsf{case}\ t\ \mathsf{of}\ \mathsf{In1}\ ec \to (\mathsf{case}\ ec\ \mathsf{of}\ \mathsf{Inl}\ e \to (\exists v.\ w = \mathsf{In1}\ v) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{Inr}\ c \to \qquad w = \bullet) \\ \qquad\qquad\qquad\qquad \mid \mathsf{In2}\ v \ \to (\exists vf.\ w = \mathsf{In2}\ vf) \\ \qquad\qquad\qquad\qquad \mid \mathsf{In3}\ es \to (\exists vs.\ w = \mathsf{In3}\ vs) \end{array}$$

Evaluation is unique, and it enjoys two propagation properties for exceptions:

$$\begin{array}{ll} unique\_eval & \Gamma\vdash\sigma \xrightarrow{t\succ} w\sigma \longrightarrow \Gamma\vdash\sigma \xrightarrow{t\succ} w\sigma' \longrightarrow w\sigma' = w\sigma \\ eval\_xcpt & \Gamma\vdash(\mathsf{Some}\ xc,s) \xrightarrow{t\succ} (w,\sigma') \longrightarrow \sigma' = (\mathsf{Some}\ xc,s) \\ eval\_no\_xcpt & \Gamma\vdash(x,s) \xrightarrow{t\succ} (w,\mathsf{Norm}\ s') \longrightarrow x = \mathsf{None} \end{array}$$

The halloc and sxalloc relations have similar properties.

Chapter 4

Type Safety

This chapter describes our soundness proof for the type system of Java^{light}. Type soundness involves both the static and the dynamic semantics as well as their interplay. Thus its proof is an excellent check for the language design itself, but also a validation aid for its formalization and a benchmark for the adequacy of the formalization for meta-theoretical proofs. If the proof fails, it should be easy to spot which of these three aspects the problem is due to.

4.1 Notions

A programming language is called *type-safe* if its design prevents type errors. A *type error* is the application of a non-function or the use of a function on arguments for which it is not defined [WF94]. Note that this general definition does not already presume the existence of a (static) type system. Type errors can have hazardous effects such as interpreting arbitrary values as pointers (and thus possibly corrupting memory) or, for object-oriented languages, calling non-existing methods. A prominent bad example has been Eiffel due to a mistake in its type system: the contravariance of method parameter types was confused with covariance [Coo89]. This failure alerted many designers of (in particular object-oriented) programming languages to take type safety extremely serious. See, for example, the series of papers by Bruce et al. [Bru93, BCM$^+$93, BGS95].

The common way to prevent type errors is to give a type system that — together with other well-formedness constraints — is strong enough to imply type safety. This property of a type system is called *type soundness*. When claiming type soundness for ML, Milner expressed it with the slogan "Well-typed expressions do not go wrong" [Mil78]. The first type soundness proofs were given in the form of so-called *subject reduction* theorems for the typed λ-calculus. The form of these theorems is "if an expression is well-typed and is reduced to some value (or some other term) then this result has a compatible type".

A programming language is *statically typed* if the absence of compile-time errors guarantees not only the absence of (run-time) type errors but also that run-time type checks are not needed. Languages like Java do not fulfill this ideal because they contain type casts and other constructs that (in general) cannot be fully checked statically. To ensure type safety, any type mismatch when evaluating problematic constructs has to be caught, and in reaction execution is stopped or a suitable exception is thrown.

In the literature the terms "type safety" and "type soundness" are sometimes used interchangeably. We will define formally only the latter term and use it in particular to stress precise technical issues. In more informal contexts we will use "type safety" even if we actually mean type soundness (implying type safety).

4.2 Relevance

Given the wide-spread use of Java programs obtained from sources that cannot be trusted (*i.e.* via the Internet), the machines executing these programs have to be protected from (intentional and non-intentional) malicious behavior. The security policy incorporated in the design of Java aiming to achieve this relies, among other things, on type-safe program execution. Special care has been taken to make the *Java Virtual Machine (JVM)* [LY96] executing compiled Java programs robust against type errors. For programs to be executed, the so-called *Bytecode Verifier* checks well-formedness, in particular well-typedness on the level of machine instructions, called *bytecode*. If the type system of the bytecode is sound, well-typedness implies the absence of runtime type mismatches. Thus type-safe execution of Java programs actually depends on two requirements:

- correct implementation of type-checking within the Bytecode Verifier. Whether this requirement is fulfilled is hard to tell as this requires program verification of complex (and typically commercial, not open-source) software.
- type soundness on the bytecode level. This has been investigated first by Qian [CGQ98, Qia99], then by Pusch [Pus98b], Nipkow [Nip00] and others.

From the security perspective, type safety is required only for the bytecode level. Type safety on the Java source level would be helpful only if one could guarantee that the bytecode to be executed has been produced with a correct compiler. Yet type soundness on the source level does have its merits:

- it is an important design check for the language. One cannot expect type soundness for the target language of compilation if it is not already given on the source level.
- it has high methodological value for program development. Since Java is a statically typed language (if we gloss over the dynamic checks for array assignments and explicit type casts), all type errors are detected already at compilation, which eases debugging and maintenance enormously.

The Java specification explicitly addresses the issue of type soundness [GJS96, §15.3]:

> The value of an expression is always assignment compatible with the type of the expression, just as the value stored in a variable is always compatible with the type of the variable. In other words, the value of an expression whose type is `T` is always suitable for assignment to a variable of type `T`.

Later in the same chapter, the reader is reminded of consequence of this general statement in the special context of method calls where `T` is the class or interface containing the method declaration as determined statically and `R` is the dynamic type of the corresponding object [GJS96, §15.11.4.4]:

> Note that for invocation mode `interface`, `R` necessarily implements `T`; for invocation mode `virtual`, `R` is necessarily either `T` or a subclass of `T`.

Clearly, in the design of Java type soundness has been an important goal. The claim quoted above shows that the designers of Java are convinced that their language is type-safe, but they do not provide any proof for this property.

4.3 Auxiliary notions

In order to express type soundness, we define a little hierarchy of auxiliary concepts of *conformance*, initially inspired by [DE97a].

Relative to a given program Γ and a state s, a value v conforms to a type T, written $\Gamma,s \vdash v{::}T$, iff the dynamic type of v widens to T:

$$\Gamma,s \vdash v{::}{\preceq}T \equiv \exists T'{\in}\mathsf{typeof}\ (\mathsf{dyn\_ty}\ s)\ v{:}\ \Gamma \vdash T' {\preceq} T$$
$$\mathsf{dyn\_ty}\ s\ a \equiv \mathsf{option\_map}\ \mathsf{obj\_ty}\ (\mathsf{heap}\ s\ a)$$

where the function `dyn_ty` calculates the dynamic type of the object at a given location on the heap. It is used by the function `typeof` in case v is an address, as defined in §2.3.

The concept of conformance for a single value extends to tables of values and their respective types, yielding a judgment of the form $prog,st \vdash (\alpha,\ val)table[{::}{\preceq}](\alpha,\ ty)table$:

$$\Gamma,s \vdash vs[{::}{\preceq}]Ts \equiv \forall n.\ \forall T{\in}Ts\ n{:}\ \exists v{\in}vs\ n{:}\ \Gamma,s \vdash v{::}{\preceq}T$$

This general notion is used for both object fields and local variables.

An object obj with reference r is conforming, expressed by a judgment of the form $prog,st \vdash obj{::}{\preceq}\surd oref$, iff the values of its fields conform to their respective types and (unless the object is a class object) its tag gives a valid dynamic type:

$$\begin{aligned}\Gamma,s \vdash obj{::}{\preceq}\surd r \equiv\ & \Gamma,s \vdash \mathsf{snd}\ obj[{::}{\preceq}]\mathsf{var\_tys}\ \Gamma\ (\mathsf{fst}\ obj)\ r\ \wedge \\ & ((\exists a.\ r = \mathsf{Heap}\ a) \longrightarrow \mathsf{is\_type}\ \Gamma\ (\mathsf{obj\_ty}\ obj))\end{aligned}$$

Note that, according to the definition of `var_tys` (*cf.* §3.1.1), the parameter r is needed to determine the fields of class objects.

Finally we can define the notion of a whole state σ conforming to an environment E. The relation $\sigma{::}{\preceq}E$ means that all objects (including class objects) as well as the current local variables in σ are conforming, and if an exception reference is present, it conforms to class `Throwable`. This implies that all values within the state are compatible with their respective static types.

$$\begin{aligned}&(x,s){::}{\preceq}(\Gamma,\ \Lambda) \equiv \\ &\quad (\forall r.\ \forall obj{\in}\mathsf{globs}\ s\ r{:}\quad \Gamma,s \vdash obj\ {::}{\preceq}\surd r)\ \wedge \\ &\quad \Gamma,s \vdash \mathsf{locals}\ s[{::}{\preceq}]\Lambda\ \wedge \\ &\quad (\forall a.\ x = \mathsf{Some}\ (\mathsf{XcptLoc}\ a) \longrightarrow \Gamma,s \vdash \mathsf{Addr}\ a\ {::}{\preceq}\mathsf{Class}\ (\mathsf{SXcpt}\ \mathtt{Throwable}))\end{aligned}$$

Note that the conformance relation is defined such that it does not take into account inaccessible values, *i.e.* values that occur in the state but not in the corresponding component of the static environment. Among others, this frees us from explicitly deleting the exception parameter from the table of local variables after a `catch` clause.

In the proofs described below we will need the property that during execution, objects are not lost and moreover retain the values of their tags. This is expressed by the following relation on states, which we call *global extension*, of the form $st{\unlhd}st$:

$$s{\unlhd}s' \equiv \forall r.\ \forall (oi,\ fs){\in}\mathsf{globs}\ s\ r{:}\ \exists (oi',fs'){\in}\mathsf{globs}\ s'\ r{:}\ oi' = oi$$

Note that if we considered garbage collection, we would have to restrict this property to accessible objects.

Each of the above notions has a number of rather straightforward properties, such as

conf_widen	$\mathsf{ws\_prog}\ \Gamma \longrightarrow \Gamma \vdash T{\preceq}T' \longrightarrow \Gamma,s \vdash v{::}{\preceq}T \longrightarrow \Gamma,s \vdash v{::}{\preceq}T'$
lconf_init_vals	$\forall n.\ \forall T{\in}fs\ n{:}\mathsf{is\_type}\ \Gamma\ T \longrightarrow \Gamma,s \vdash \mathsf{init\_vals}\ fs[{::}{\preceq}]fs$
oconf_cong	$\Gamma,\mathsf{set\_locals}\ l\ s \vdash obj{::}{\preceq}\surd r = \Gamma,s \vdash obj{::}{\preceq}\surd r$
conforms_xgext	$(x\ ,s){::}{\preceq}(\Gamma,\Lambda) \wedge (x',\ s'){::}{\preceq}(\Gamma,\ \Lambda) \wedge s'{\unlhd}s \longrightarrow (x',s){::}{\preceq}(\Gamma,\Lambda)$
inited_gext	$s{\unlhd}s' \longrightarrow \mathsf{inited}\ C\ (\mathsf{globs}\ s) \longrightarrow \mathsf{inited}\ C\ (\mathsf{globs}\ s')$

4.4 Goal

With the help of the notions just introduced we can express our goal of type soundness as follows. In the context of a well-formed program, if the execution of a well-typed statement c transforms a state conforming to the environment into another state then that state again conforms to the environment:

$$\mathsf{wf\_prog}\ \Gamma \wedge (\Gamma,\Lambda)\vdash c::\surd \wedge \sigma::\preceq(\Gamma,\Lambda) \wedge \Gamma\vdash \sigma \xrightarrow{c} \sigma' \longrightarrow \sigma'::\preceq(\Gamma,\Lambda)$$

Analogously, the evaluation of a well-typed expression e preserves the conformance of the state to the environment. In addition, unless an exception has occurred, the result of the expression conforms to its static type:

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \wedge (\Gamma,\Lambda)\vdash e::T \wedge \sigma::\preceq(\Gamma,\Lambda) \wedge \Gamma\vdash \sigma \xrightarrow{e\succ v} (x',s') \longrightarrow \\ \quad (x',s')::\preceq(\Gamma,\Lambda) \wedge (x' = \mathsf{None} \longrightarrow \Gamma,s'\vdash v::\preceq T)\end{array}$$

We will further need similar properties for variables and expressions. Thanks to the uniformity of our well-typedness and evaluation judgments, all parts of the resulting four formulas can be shared if we further introduce a uniform notion of result conformance given below. Thus the actual main proof obligation, with explicit universal quantification where needed in order to obtain a strong enough induction hypotheses for rule induction, reads as

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \longrightarrow \Gamma\vdash \sigma \xrightarrow{t\succ} (v,\sigma') \longrightarrow \forall\Lambda.\ \sigma::\preceq(\Gamma,\Lambda) \longrightarrow \\ \quad \forall T.\ (\Gamma,\Lambda)\vdash t::T \longrightarrow \sigma'::\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma' = \mathsf{None} \longrightarrow \Gamma,\Lambda,\mathsf{snd}\ \sigma'\vdash t\succ v::\preceq T)\end{array}$$

where

$$\begin{array}{l}\Gamma,\Lambda,s\vdash \mathsf{In1}\ \_\succ \mathsf{In1}\ v\ \ ::\preceq\mathsf{Inl}\ T\ =\ \Gamma,s\vdash v::\preceq T \\ \Gamma,\Lambda,s\vdash \mathsf{In2}\ \_\succ \mathsf{In2}\ (v,f)::\preceq\mathsf{Inl}\ T\ = (\Gamma,s\vdash v::\preceq T \wedge s\trianglelefteq f\preceq T::\preceq(\Gamma,\Lambda)) \\ \Gamma,\Lambda,s\vdash \mathsf{In3}\ \_\succ \mathsf{In3}\ vs\ \ ::\preceq\mathsf{Inr}\ Ts = \mathsf{list\_all2}\ (\lambda v\ T.\ \Gamma,s\vdash v::\preceq T)\ vs\ Ts \\ s\trianglelefteq f\preceq T::\preceq(\Gamma,\Lambda) \equiv \forall s'\ w.\ \mathsf{Norm}\ s'::\preceq(\Gamma,\Lambda) \longrightarrow \Gamma,s'\vdash w::\preceq T \longrightarrow s\trianglelefteq s' \longrightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{assign}\ f\ w\ (\mathsf{Norm}\ s')::\preceq(\Gamma,\Lambda)\end{array}$$

In the result conformance predicate, which is of the form $prog,lenv,\mathsf{st}\vdash term\succ vals::\preceq tys$, the term argument determines which notion of conformance is actually used. The second case is more involved than the others: for a variable we have to state not only that its value conforms to the variable type but also that assigning to it a new value (conforming to the same type) in any state $\mathsf{Norm}\ s'$ that is an extension of the current state preserves conformance.

Of course, the two properties given above as our initial goals now result as corollaries from the main theorem. A further interesting corollary is that method calls with dynamic binding always execute a suitable method, *i.e.* 'method not understood' run-time errors are impossible. More formally, for a well-formed program and a state σ that conforms to the current environment, if a method call $e..mn(ps)$ with invocation mode "Interface or Virtual" is well-typed and e evaluates from σ without an exception to a non-null value then this value is an address of an existing object obj and the method lookup for the given signature $(mn,\ pTs')$ within the dynamic type of obj yields a proper method body:

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma \wedge (\Gamma,\Lambda)\vdash \{t,md,\mathsf{IntVir}\}e..mn(\{pTs'\}ps)::-rT \wedge \sigma::\preceq(\Gamma,\Lambda) \wedge \\ \Gamma\vdash \sigma \xrightarrow{e-\succ a'} \mathsf{Norm}\ s' \wedge a' \neq \mathsf{Null} \longrightarrow \\ \quad \exists a\ obj.\ a' = \mathsf{Addr}\ a \wedge \mathsf{heap}\ s'\ a = \mathsf{Some}\ obj \wedge \mathsf{cmethd}\ \Gamma\ (\mathsf{obj\_class}\ obj)\ (mn,\ pTs') \neq \mathsf{None}\end{array}$$

4.5 Proof

We have to prove both the global extension property and actual type soundness for all kinds of evaluation relations as well as the auxiliary relations halloc and sxalloc. Fortunately, global extension can be proved in advance, *i.e.* independently from type soundness. Since sxalloc depends on halloc and the four kinds of evaluation relations depend on each other and the two auxiliary relations, we have to perform the proofs in the order given below.

We prove (by rule induction) global extension for halloc, sxalloc, and then for the remaining relations:

$$\begin{array}{ll}
\textit{halloc_gext} & \Gamma\vdash\sigma_1 \xrightarrow{\mathsf{halloc}\ oi\succ a} \sigma_2 \longrightarrow \mathsf{snd}\ \sigma_1 \trianglelefteq \mathsf{snd}\ \sigma_2 \\
\textit{sxalloc_gext} & \Gamma\vdash\sigma_1 \xrightarrow{\mathsf{sxalloc}} \sigma_2 \longrightarrow \mathsf{snd}\ \sigma_1 \trianglelefteq \mathsf{snd}\ \sigma_2 \\
\textit{eval_gext_lemma} & \Gamma\vdash\sigma \xrightarrow{t\succ} (w,\sigma') \longrightarrow \mathsf{snd}\ \sigma \trianglelefteq \mathsf{snd}\ \sigma' \wedge (\mathsf{case}\ w\ \mathsf{of} \\
& \quad \mathsf{In1}\ v \to \mathsf{True} \\
& \quad \mid \mathsf{In2}\ vf \to \mathsf{normal}\ \sigma \longrightarrow (\forall v\ x\ s.\ s \trianglelefteq \mathsf{snd}\ (\mathsf{assign}\ (\mathsf{snd}\ vf)\ v\ (x,s))) \\
& \quad \mid \mathsf{In3}\ vs \to \mathsf{True})
\end{array}$$

Then we prove (some variants of) type soundness for the two auxiliary relations.

$$\begin{array}{ll}
\textit{halloc_ts} & \mathsf{wf_prog}\ \Gamma \wedge \Gamma\vdash\sigma_1 \xrightarrow{\mathsf{halloc}\ oi\succ a} \sigma_2 \wedge \sigma_1{::}\preceq(\Gamma,\Lambda) \wedge \mathsf{is_type}\ \Gamma\ (\mathsf{obj_ty}\ (oi,fs)) \longrightarrow \\
& \sigma_2{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_2 = \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_2\vdash\mathsf{Addr}\ a{::}\preceq\mathsf{obj_ty}\ (oi,fs)) \\
\textit{sxalloc_ts} & \mathsf{wf_prog}\ \Gamma \wedge \Gamma\vdash\sigma_1 \xrightarrow{\mathsf{sxalloc}} \sigma_2 \longrightarrow \mathsf{case\ fst}\ \sigma_1\ \mathsf{of\ None} \to \sigma_2 = \sigma_1 \\
& \mid \mathsf{Some}\ x \to \exists a.\ \mathsf{fst}\ \sigma_2 = \mathsf{Some}\ (\mathsf{XcptLoc}\ a) \wedge (\forall\Lambda.\ \sigma_1{::}\preceq(\Gamma,\Lambda) \longrightarrow \sigma_2{::}\preceq(\Gamma,\Lambda))
\end{array}$$

Finally, we prove the main type soundness theorem by rule induction on the evaluation relations. The proof consists of one case per syntactic construct, currently 27.

- 12 cases can be solved rather directly (e.g. from the induction hypothesis), like the expression statement Expr e:
$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \Gamma\vdash\mathsf{Norm}\ s_0 \xrightarrow{e-\succ v} \sigma_1 \wedge \\
(\forall\Lambda.\ \mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \longrightarrow (\forall T.\ (\Gamma,\Lambda)\vdash e{::}{-}T \longrightarrow \\
\quad \sigma_1{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_1 = \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_1\vdash v{::}\preceq T))) \wedge \\
\mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \wedge (\Gamma,\Lambda)\vdash e{::}{-}T \longrightarrow \\
\quad \sigma_1{::}\preceq(\Gamma,\Lambda)
\end{array}$$
- 9 cases require just a few lemmas on the structure of the state, like the class instance creation `new` C:
$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \Gamma\vdash\mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ C} \sigma_1 \wedge \\
(\forall\Lambda.\ \mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \longrightarrow \mathsf{is_class}\ \Gamma\ C \longrightarrow \sigma_1{::}\preceq(\Gamma,\Lambda)) \wedge \\
\Gamma\vdash\sigma_1 \xrightarrow{\mathsf{halloc\ CInst}\ C\succ a} \sigma_2 \wedge \mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \wedge \mathsf{is_class}\ \Gamma\ C \longrightarrow \\
\quad \sigma_2{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_2\ \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_2\vdash\mathsf{Addr}\ a{::}\preceq\mathsf{Class}\ C)
\end{array}$$
- the remaining 6 cases require extensive reasoning on constructs involved: *viz.* class initialization, assignment, `try _ catch`, field and array variables, and the method call $\{t,cT,mode\}e..mn(\{pTs\}ps)$ as the most complex one:
$$\begin{array}{l}
\mathsf{wf_prog}\ \Gamma \wedge \Gamma\vdash\mathsf{Norm}\ s_0 \xrightarrow{e-\succ a'} \sigma_1 \wedge \\
(\forall\Lambda.\ \mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \longrightarrow (\forall T.\ (\Gamma,\Lambda)\vdash e{::}{-}T \longrightarrow \\
\quad \sigma_1{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_1 = \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_1\vdash a'{::}\preceq T))) \wedge \\
\Gamma\vdash\sigma_1 \xrightarrow{ps\doteq\succ vs} \sigma_2 \wedge \\
(\forall\Lambda.\ \sigma_1{::}\preceq(\Gamma,\Lambda) \longrightarrow (\forall Ts.\ (\Gamma,\Lambda)\vdash ps{::}\doteq Ts \longrightarrow \\
\quad \sigma_2{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_2 = \mathsf{None} \longrightarrow \mathsf{list_all2}\ (\lambda v\ T.\ \Gamma,\mathsf{snd}\ \sigma_2\vdash v{::}\preceq T)\ pvs\ Ts))) \wedge \\
C = \mathsf{target}\ (\mathsf{invmode}\ m\ e)\ (\mathsf{snd}\ \sigma_2)\ a'\ cT \wedge \\
\sigma_2' = \mathsf{init_lvars}\ \Gamma\ C\ (mn,\ pTs)\ (\mathsf{invmode}\ m\ e)\ a'\ pvs\ \sigma_2 \\
\Gamma\vdash\sigma_2' \xrightarrow{\mathsf{Methd}\ C\ (mn,\ pTs)-\succ v} \sigma_3 \wedge \\
(\forall\Lambda.\ \sigma_2'{::}\preceq(\Gamma,\Lambda) \longrightarrow (\forall T.\ (\Gamma,\Lambda)\vdash\mathsf{Methd}\ C\ (mn,\ pTs){::}{-}T \longrightarrow \\
\quad \sigma_3{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_3 = \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_3\vdash v{::}\preceq T))) \wedge \\
\mathsf{Norm}\ s_0{::}\preceq(\Gamma,\Lambda) \wedge (\Gamma,\Lambda)\vdash e{::}{-}\mathsf{RefT}\ t \wedge (\Gamma,\Lambda)\vdash ps{::}\doteq pTsa \wedge \\
\mathsf{max_spec}\ \Gamma\ t\ (mn,\ pTsa) = \{((cT,\ m,\ pns,\ rT),\ pTs)\} \longrightarrow \\
\quad \mathsf{restore_lvars}\ \sigma_2\ \sigma_3{::}\preceq(\Gamma,\Lambda) \wedge (\mathsf{fst}\ \sigma_3 = \mathsf{None} \longrightarrow \Gamma,\mathsf{snd}\ \sigma_3\vdash v{::}\preceq rT)
\end{array}$$

Factoring out the reasoning for the more complex cases into one or several lemmas helps to keep the main proof manageable.

4.6 Discussion

This section comments on a potential weakness of our approach using an evaluation semantics as the basis for the proof of type soundness. We sketch the alternative using a transition semantics. In the subsequent section, we point out the problems of that approach, as originally presented in [Ohe98].

4.6.1 Non-termination

As it stands, our type soundness theorem does not directly say anything about non-terminating computations, which might lead to the conclusion that it is useless for the type-safe execution of reactive systems and other looping programs. Fortunately, the theorem carries over even to such cases if one accepts the following meta-level reasoning:

> Since every infinite computation consists of more than one statement, each non-terminating program can be interrupted after any finite number of statements executed, for example by introducing a counter and raising an exception when a given number of statements has been reached. The theorem implies that the state resulting from interrupting the computation after any finite number of statements executed conforms to the environment. Thus the whole (infinite) computation can be concluded to be type-safe.

4.6.2 Alternative: Transition Semantics

A more natural account for non-termination could be given with a subject reduction theorem for a transition semantics rather than evaluation semantics. In this case the main theorem would look like

$$\begin{array}{l}\mathsf{wf\_prog}\ \Gamma\ \wedge\ (\Gamma,\Lambda),\mathsf{dyn\_ty}\ (\mathsf{snd}\ \sigma)\models t{::}\,T\ \wedge\ \sigma{::}\preceq(\Gamma,\Lambda)\ \wedge\ \Gamma\vdash(t,\sigma)\rightarrow_1(t',\sigma')\ \longrightarrow\\ \quad\sigma'{::}\preceq(\Gamma,\Lambda)\ \wedge\ (\mathsf{fst}\ \sigma' = \mathsf{None}\longrightarrow\exists T'.\ (\Gamma,\Lambda),\mathsf{dyn\_ty}\ (\mathsf{snd}\ \sigma')\models t{::}\,T'\wedge\Gamma\vdash T'\preceq T)\end{array}$$

where $\_,\_\models\_{::}\_$ is the extended typing judgment taking into account the dynamic types of objects occurring in partially evaluated terms (*cf.* §2.8.2), and $\Gamma\vdash(t,\sigma)\rightarrow_1(t',\sigma')$ should mean that the configuration consisting of a term t and a state σ is transformed (in an atomic step) to (t',σ').

Together with the progress property $\neg\mathsf{ground}\ t\ \longrightarrow\ \exists t'\ \sigma'.\ \Gamma\vdash(t,\sigma)\rightarrow_1(t',\sigma')$, which has to be proved *e.g.* by structural induction, type safety for both finite and infinite computations is obvious.

The above subject reduction theorem can again be proved by rule induction. However, this proof is much more involved than for an evaluation semantics. Next to the general drawbacks already mentioned in §3.2.1, new difficulties arise, namely problematic intermediate situations during program execution, as described in the next section.

4.7 Problems with Transition Semantics

When Drossopoulou and Eisenbach did their first version of their type soundness proof for a small subset of Java [DE97a], and when we shortly after that decided to prefer a transition semantics over an evaluation semantics, apparently no one was fully aware of the trouble transition semantics brings to the proof of type soundness. When adding arrays to their sublanguage [DE97b], they recognized a phenomenon that we call the *array problem.* A related problem which we call *conditional problem* was later brought up by Hosoya, Pierce and Turner and has been discussed in an e-mail forum [PHT+98].

4.7.1 Problem Origins

Essentially, both problems mentioned above are due to the fact that the normal typing rules correspond to the final outcome of terms, which fits perfectly with an evaluation semantics but not necessarily fits with a transition semantics with its intermediate states. In a transition semantics, addresses (*i.e.* literal values of the form Lit (Addr a)) emerge as intermediate expressions, and due to subtyping their types may be narrower than the types of the original (unreduced) expressions. This leads to ill-typed intermediate expressions in at least two cases.

4.7.2 Array Problem

Consider an array variable `arr` that is statically of some type C[] and dynamically of some subtype D[]. Assigning to one of its elements an expression e that is statically of type C and dynamically (*i.e.* after full evaluation) of type D is perfectly legal and type-safe. Yet when executing the assignment expression `arr`[i]`:=`e stepwise we get to an intermediate expression Lit (Addr a)[i]`:=`e that is ill-typed because Lit (Addr a) is already of type D[] and the right-hand side e is still of the (super)type C. Only after evaluating also e well-typedness is restored.

Essentially the same problem arises when the left-hand side gets narrower, as before, but the right-hand side remains of type C. Surely, an `ArrStore` exception is thrown immediately after both sides have been evaluated, but the intermediate expression where just the left-hand side has been evaluated is ill-typed.

The problem could be circumvented by evaluating the right-hand side before the left-hand side, but thus would alter the semantics of the language, which is not acceptable. It would disappear also if Java arrays were non-variant, *i.e.* did not allow subtyping on arrays at all. Drossopoulou and Eisenbach as well as Syme solve the problem as follows [DE99, Sym99b]. They relax the typing rule for array assignments by removing the widening requirement between the right-hand and left-hand side. On the other hand they have to prove the so-called *array property* stating that the dynamic type check performed for array assignments is sufficient to preserve conformance.

4.7.3 Conditional Problem

Consider two parameters a and b (of some method m) whose types are A and B where A widens to B or vice versa. Thus the conditional expression c `?` a `:` b within m is well-typed and its type is the narrower one of A and B. Assume further that m is called with actual arguments whose types are some subtype C of A and some subtype D of B, respectively, where C and D themselves are not in widening relation. Up to now, everything is fine since the method call is also well-typed. Yet when executing the call, inserting the argument values, the types of a and b become C and D, respectively, and c `?` a `:` b is no longer well-typed because neither C widens to D nor vice versa. Note that this problem necessarily involves method calls because this is the only situation where both branches of the conditional expression are evaluated in advance (due to call-by-value semantics). Otherwise, first c is evaluated and then either a or b is selected for further reduction whereby the conditional expression disappears.

Several solutions to this problem have been proposed [PHT+98], namely

- adding type casts or intermediate variables, acting as annotations of the intended (original) types, for both branches of the conditional expression,
- adding a type annotation for the result of the conditional expression, which relieves from inferring that type,

- relaxing (an intermediate form of) the typing rule such that it allows for any common supertype of the two argument types as the result type, which unfortunately makes typing non-unique,
- inventing joins[1] (*i.e.* least upper bounds) of arbitrary reference types and take as the result type the join of both argument types.

All solutions change the language or at least require an intermediate language with an adapted type system, where the amount of change increases from almost negligible (for the first solution) to heavy (for the last one).

Our contribution to the discussion were clarifications and summarization. The reason why we consider in Java^{light} the otherwise uninteresting conditional expression at all in our model was to demonstrate that at least for an evaluation semantics it does not cause any problems. It was only recently when we noticed that also for a transition semantics the conditional problem occurs only if method calls are inlined (*i.e.* β-reduced) such that the actual argument values are substituted for the formal parameters within the method body.

In the discussion, both Drossopoulou and Syme promoted the third of the above solutions for use with their model. Yet their model does not at all suffer from the problem because it performs explicit substitution: it stores the argument values in the state as the values of fresh variables, and only the names of the new variables are substituted for the parameter names in the method body. As a consequence, the typing of the parameters (according to the environment) is maintained.

4.7.4 Side Effects on Types

When Syme machine-checked the proof of Drossopoulou and Eisenbach, he found — and corrected — in their proofs a subtle error related to the array problem [Sym99b, §6.2]. Furthermore, he recognized a major omission concerning side effects on types [Sym99b, §6.3]: reducing a term might affect the types of other terms (*e.g.* further subterms of a common superterm) due to side-effects on the state. Consequently, he requires and proves a lemma stating that execution preserves typing up to widening. Drossopoulou and Eisenbach take into account the improvements pointed out by Syme in later versions of their proof [DE99, §9.2] and show that even strict type preservation holds: the types of other terms are not affected at all.

4.8 Summary

We have proven

Theorem 1 *Java, as far as covered by our model, is type-safe.*

Once the rather large number of required concepts and properties are identified, for an evaluation semantics the proof is moderately difficult, yet lengthy. The type safety result directly applies to terminating evaluations and can be carried over also to infinite computations. For a transition semantics, the proof would be even more involved.

[1] See for example the work by Büchi and Weck [BW98] for the benefits and an implementation of compound types for Java.

Chapter 5

Axiomatic Semantics

In this chapter we present a Hoare-style logic that is an important first step towards verifying Java programs. Here we do not focus on the methodological issues involved with program verification as done *e.g.* by Poetzsch-Heffter and Müller [PHM99, MPH99], but concentrate on the meta-theory and its mechanization.

We describe how to extend classical Hoare logic [Hoa69, Apt81] to handle side-effecting expressions, intermediate values, exceptions, mutual recursion, dynamic binding, static initialization and other difficult features. We first give our axiomatic semantics of partial correctness, motivating and discussing all required concepts like assertions and validity. Then we sketch our proof of soundness and (relative) completeness, both w.r.t. a slight elaboration of the operational semantics given in §3.

More or less self-contained articles covering the topic of this chapter are [Ohe00a], [Ohe00b] and [Ohe01].

5.1 Assertions

In designing an axiomatic semantics the most critical notion is that of *assertions*, *i.e.* propositions describing the pre- and postconditions of term execution. The language of assertions and the underlying logic strongly determine the expressiveness and completeness of the resulting verification logic.

5.1.1 Logical Language

As the assertion language and logic, we could use *Peano Arithmetic*, *i.e.* first-order predicate logic with equality, natural numbers, $+$ and $*$. This is because we will quantify essentially just over (lists and finite mappings of) values. The program state can be encoded using lists of (lists of) values, the potentially problematic variable update functions (contained in type *vval*) can be coded as a simple choice between the three possible cases lvar, fvar and avar, and for lists standard encodings exist. Thus a rather minimal language would be sufficient, but at the expense of technically awkward encodings. Based on these observations, we could define a notion of *expressiveness* suiting our needs.

Moreover, we could embed the assertion language, being higher-order or not, deeply into HOL. Doing so would enable us to explicitly treat expressiveness and the whole issue of completeness (basically) along the lines of Cook [Coo78], who did a good job separating concerns within the completeness affair. Yet as observed by Kleymann [Kle98, §2.12], some of the known incompleteness results crucially depend on certain expressiveness properties and have been misinterpreted in the sense that they were attributed relevance for practical purposes, which they in fact do not have. Actually, we are not aware of any actual verification system based on Hoare logic where incompleteness of the underlying logic is an issue. Thus Kleymann and others follow Aczel's suggestion [Acz82] not to consider expressiveness when investigating the completeness of a Hoare-style logic. Moreover, compared to a shallow embedding, a deep embedding would complicate in particular the meta-level proofs as it requires talking explicitly about the syntactic level (*i.e.* terms and substitutions) and their semantic interpretation, which would just add clutter without giving us any benefit.

For the reasons given above we shallow-embed assertions in HOL. As a result, we do not have to bother with expressiveness. Concerning derivability, we automatically only have to deal with *relative completeness* in the sense of Cook [Coo78], *i.e.* completeness of the Hoare logic rule system itself modulo (in-)completeness of the underlying meta logic. In particular, within the rule of consequence, the derivability of an implication between assertions is replaced simply by its validity, *i.e.* mere implication in the meta logic HOL.

Since assertions depend on the program variables (including the heap), from the HOL perspective, they are essentially just predicates on the state. We use the state as an explicit parameter of the assertions, which is most appropriate when conducting meta-theory. Thus an example Hoare triple that is traditionally given as $\{\mathsf{True}\}\ c\ \{X{=}1\}$ now formally reads as $\{\lambda\sigma.\ \mathsf{True}\}\ c\ \{\lambda\sigma.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = \mathsf{Some}\ (\mathsf{Intg}\ 1)\}$, where the rather cumbersome expression in the postcondition could of course be suitably abbreviated and the assertions pretty-printed. For actual program verification, a mechanism for hiding the state and directly referring to the program variable names, as given by Wenzel [Wen00], would enhance readability.

5.1.2 Auxiliary Variables

Program verification typically involves relating pre- and postconditions of program terms, in particular when stating that a certain portion of the state does not change or when giving input-/output specifications of methods (or general procedures). Such relations are easily expressed in VDM [Jon90] using "hooked" expressions within the postcondition to refer to the initial state. With plain Hoare logic, one cannot make such references, but one can extend the logic by so-called *auxiliary variables*, or *logical variables*, which are universally bound at a higher level.

For example, the proposition that a procedure P does not change the contents of a program variable X may be formulated as the triple $\{X{=}Z\}\ \mathsf{Call}\ P\ \{X{=}Z\}$, which should mean that whenever X has some value denoted by Z before calling P, after return it still has the same value, as given by Z. A potential interpretation for Z is to be a program variable not occurring in P, but this essential side condition cannot be expressed within the logic. A better and rather intuitive alternative is that Z be viewed as a free variable, which is thus implicitly universally quantified at the outermost logical level. Yet this gives the desired interpretation only if the triple occurs (implication-)positively, and thus is unsuitable when triples are used also in assumptions as required for verifying recursive methods (*cf.* §5.6.8).

Viewing Z as an arbitrary (yet fixed) constant preserves correctness, but this approach suffers from incompleteness: take a procedure triple like $\{X{=}Z\}\ \mathsf{Call}\ \mathit{Square}\ \{Y{=}Z{*}Z\}$ as an example. Both for handling recursive calls during its proof and for different applications after it has been proved, different instantiations of Z may be required, which is impossible if Z is a constant. The classical but cumbersome and often incomplete way out, as described *e.g.* by Apt [Apt81], is inventing some (more or less ad-hoc) set of substitution and adaptation rules involving sometimes intricate side-conditions on variable occurrences. A semantically satisfactory solution that we could adopt would involve (implicit or explicit) universal quantification at the level of triples like $\forall Z.\ \{P\ Z\}\ c\ \{Q\ Z\}$, but this changes the outer structure of Hoare triples and makes them more difficult to handle, in particular if they occur in assumptions.

We follow the approach promoted and applied by Kleymann [Sch97]: implicit quantification of auxiliary variables at the level of triple validity. In order to abstract from the number and the names of the auxiliary variables used in different triples, assertions receive an extra parameter representing the collection of all required auxiliary variables. The type of the parameter is not specified and thus can be instantiated as appropriate, typically to a tuple of values (for actual program verification) or the whole state (for meta theory). With this extended notion of assertions, the motivating example $\{X{=}Z\}$ Call P $\{X{=}Z\}$ now reads as $\{\lambda\sigma\ Z.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = Z\}$ Call P $\{\lambda\sigma\ Z.\ \mathsf{locals}\ (\mathsf{snd}\ \sigma)\ X = Z\}$ where in this case the type of Z is of course the type of the variable X.

5.1.3 Result Values

We still need a further — orthogonal — extension of assertions, namely for handling the *result values* of side-effecting expressions. In contrast to most other axiomatic semantics given in the literature, and as already motivated in §2.4.2, we take such expressions seriously.

Homeier and Martin [HM95] appear to be the first and only ones so far embedding side-effecting expressions in a machine-checked axiomatic semantics. They transform expressions syntactically into the assertion language while using simultaneous substitutions to account for side-effects. This solution does not require special triples and result value entries. On the other hand, it is not general enough because it can handle only variable assignments (including *e.g.* incrementation operators) but not method calls within expressions which appear frequently in object-oriented programs.

We use triples not only to describe the behavior of statements, but also all other classes of terms, *i.e.* expressions, expression lists, and variables. This requires a mechanism for not only recording, but also for passing on the values produced by these terms. In an operational semantics, the (nameless) result values can be referred to and passed on via meta variables bound at the outermost logical level, but in an axiomatic semantics, such a simple technique is impossible: the behavior of a term has to be described solely by a suitable triple without any reference to its surroundings. Thus all variables occurring in the pre- and postconditions of the triple have to be logically bound to that triple. Violating this principle easily leads to unsound or incomplete rule systems.

Kowaltowski [Kow77] rather early pointed to the right direction giving a surprisingly simple (syntactic) solution: Within assertions there is a default reference, call it ρ, to the result of the current expression. The rule for constant expressions, for example, then reads as

$$\frac{}{\{P[c/\rho]\}\ c\ \{P\}}$$

which is reminiscent of the well-known assignment rule applied to ρ:=c. The rule for an arbitrary binary operator $*$ reads as

$$\frac{\{P\}\ d\ \{Q[\rho/\tau]\}\quad \{Q\}\ e\ \{R[\tau{*}\rho/\rho]\}}{\{P\}\ d{*}e\ \{R\}}\quad \tau \text{ does not occur elsewhere}$$

and can be justified by simulating the the special result variables with intermediate program variables. Unfortunately these rules cannot be used directly in our rigorous semantical setting as they rely on syntactic substitutions and the problematic syntactic side condition of variable freshness.

After some experimentation we found and implemented a first solution of the result representation problem: let the assertions refer to a stack of result values. This not only gives a default reference, *viz.* the stack top, but also an arbitrary number of unique references for further intermediate results where the explicit syntactic shifting performed when having two or more intermediate results is handled by pushing and popping elements.

Later we noticed that there is a simpler solution: assertions receive the (single) current result value as a parameter. Thus substitution can be modeled simply by a combined abstraction and application, and the rule for constant expressions reads as

$$\frac{}{\{\lambda\rho.\ P\ c\}\ c\ \{P\}}$$

Note that the precondition effectively ignores the result parameter ρ, *i.e.* $P\ c$ does not depend on it, since it makes sense only in the postcondition. We will later (*cf.* §5.1.5) abbreviate the precondition to $P{\leftarrow}c$.

Multiple result values within a Hoare logic rule can be handled by suitable explicit universal quantification and substitution of all but the last value in the following way. First, observe that we may rewrite the rule for binary operators to

$$\frac{\{P\}\ d\ \{Q\} \quad \{Q[\tau/\rho]\}\ e\ \{R[\tau{*}\rho/\rho]\}}{\{P\}\ d{*}e\ \{R\}} \quad \text{for some } \tau \text{ not occurring elsewhere}$$

This form has two advantages: we only need substitution to ρ and the side condition on τ can be made local to the second triple in the assumptions. Thus we can model the side condition semantically by universal quantification of τ around the second triple and end up with the rule

$$\frac{\{P\}\ d\ \{Q\} \quad \forall\tau.\ \{\lambda\rho.\ Q\ \tau\}\ e\ \{\lambda\rho.\ R\ (\tau{*}\rho)\}}{\{P\}\ d{*}e\ \{R\}}$$

Both subexpressions are evaluated in sequence, where Q as intermediate assertion typically involves the result of d. The final postcondition R is modified for the proof on e as follows: we take the second intermediate result ρ, combine it with the first intermediate result τ as obtained from the precondition, and use the combined value as the overall result.

We define the type *res* of the result parameter simply as the generalized result type given already for the operational semantics (*cf.* §3.2.2)

$$res = vals$$

and will use the abbreviations

$$\begin{aligned}
&\mathsf{Val}\ \ v \equiv \mathsf{In1}\ v\\
&\mathsf{Var}\ \ v \equiv \mathsf{In2}\ v\\
&\mathsf{Vals}\ v \equiv \mathsf{In3}\ v
\end{aligned}$$

for injecting single values, variables and value lists into *res*.

The names Val, Var, and Vals will be used not only as constructors, but also as (destructor) patterns. For example, $\lambda\mathsf{Val}\ v.\ f\ v$ is a function on the result entry that expects a single value v and passes it to f.

5.1.4 Assertion Type

Having decided on the logical language, the use of auxiliary variables, and the result entry, we can finally give the type of assertions. It has a type parameter α for the auxiliary variables and is defined as a relation between *res*, *state* and the auxiliary variables:

$$\alpha\ assn = res \to state \to \alpha \to bool$$

See §6.3.4 for application examples.

For implications between assertions we use the abbreviation

$$P \to Q \equiv \forall Y\, \sigma\, Z.\ P\ Y\, \sigma\ Z \longrightarrow Q\ Y\, \sigma\ Z$$

As done here, we typically refer to the result parameter of an assertion as Y, the state as σ, and the auxiliary variables as Z.

5.1.5 Combinators

In order to keep the axiomatic rules short and thus more readable, we define several assertion (predicate) combinators hiding the state, result, and auxiliary variables in applications as far as possible.

- $\lambda s..\ P\ s \equiv \lambda Y\, \sigma.\ P\ (\mathsf{snd}\ \sigma)\ Y\, \sigma$ allows P to peek at the state (without the exception status) directly.
- $P \wedge.\ p \ \equiv \lambda Y\, \sigma\ Z.\ P\ Y\, \sigma\ Z \wedge p\ \sigma$ means that not only P holds but also p, applied to the program state only. The assertion

  $\mathsf{Normal}\ P \equiv P \wedge.\ \mathsf{normal}$

  is a simple application stating that P holds and no exception has occurred.
- $f\ .;\ P \equiv \lambda Y\, \sigma.\ P\ Y\ (f\ \sigma)$ means that P holds for the state transformed by f.
- $P\ ;.\ f \equiv \lambda Y\, \sigma'\ Z.\ \exists \sigma.\ P\ Y\, \sigma\ Z \wedge \sigma' = f\ \sigma$ means that P held for some state σ and the current state is the image of σ under the state transformer f.

Note that the latter two combinators have (almost) inverse effect, in the following sense: $((f\ .;\ P)\ ;.\ f) \to P$ and $P \to (f\ .;\ (P\ ;.\ f))$.

Another group of combinators provides abbreviations for producing and consuming results:

- $P{\leftarrow}w \equiv \lambda Y.\ P\ w$ means that P holds where w is substituted as the result.
- $\lambda w :.\ P\ w \equiv \lambda Y.\ P\ Y\ Y$ peeks at the current result and passes it to $P :: \mathit{res} \to \alpha\ \mathit{assn}$.
- $P{\downarrow} \equiv \lambda Y\, \sigma\ Z.\ \exists Y.\ P\ Y\, \sigma\ Z$ simply ignores the result.
- $P{\downarrow}{=}w \ \equiv \lambda Y :.\ P{\downarrow} \wedge.\ (\lambda s.\ Y{=}w)$ asserts that the current result is w and then ignores it.

For successive substitutions the leftmost one prevails: $P{\leftarrow}w{\leftarrow}v = P{\leftarrow}w$.

5.2 Triples

We define Hoare-style *triples* (as usual consisting of a term and two assertions as pre- and postconditions) via a datatype

$(\alpha)\mathit{triple}$

with a single constructor with the mixfix syntax:

$\{\alpha\ \mathit{assn}\}\ \mathit{terms}{\succ}\ \{\alpha\ \mathit{assn}\}$

We give — again in analogy to typing and evaluation judgments — a variant for each class of terms:

$\{P\}\ e\!-\!\succ\ \{Q\} \equiv \{P\}\ \mathsf{In1l}\ e\!\succ\ \{Q\}$
$\{P\}\ e\!\!=\!\!\succ\ \{Q\} \equiv \{P\}\ \mathsf{In2}\ \ e\!\succ\ \{Q\}$
$\{P\}\ e\dot{=}\!\succ\ \{Q\} \equiv \{P\}\ \mathsf{In3}\ \ e\!\succ\ \{Q\}$
$\{P\}\ \ .c.\ \ \{Q\} \equiv \{P\}\ \mathsf{In1r}\ c\!\succ\ \{Q\}$

In some triples of our Hoare logic rules given below the term parameter will be a (quantifier-free) meta-level expression such as $\mathsf{if}\ b\ \mathsf{then\ Skip\ else}\ c$ rather than a pure term of (the abstract syntax of) Java$^{\ell ight}$. This does not hinder the use of such rules since during application these expressions are reduced to pure Java$^{\ell ight}$ terms, in this case to either Skip or c. Another typical example is the expression $\mathsf{body}\ \Gamma\ C\ sig$ which is ultimately replaced by the actual body of the given method.

For handling recursive methods we take (a variant of) the standard approach where triples appear not only in the conclusion: sets of triples are used as assumptions (*i.e.* contexts) within the validity and derivability judgments.

$(\alpha)\mathit{triples} = ((\alpha)\mathit{triple})\mathit{set}$

Furthermore, in order to handle mutual recursion, it is convenient to use sets of triples as (multiple) conclusions as well, as further explained in §5.6.8. Semantically speaking, forming sets of triples always means conjunction of its members. Note that for simplicity we allow infinite sets here though only finite sets of triples are derivable.

Actually, the triple type should not have a type parameter and triples should be universally quantified over the type of the auxiliary variables instead: $\forall\alpha.\ \{\alpha\ \mathit{assn}\}\ \mathit{terms}\!\succ\ \{\alpha\ \mathit{assn}\}$. This is not possible in HOL due to the weak (parametric) polymorphism. As a consequence, all members of a set of triples and also the assumptions and conclusions as a whole, and thus all triples within a single derivation, are bound to have the same type of auxiliary variables.

5.3 Validity

5.3.1 Single Triples

The *validity* of a single triple is a judgment of the form

$\mathit{prog}\models \mathit{nat}\colon(\alpha)\mathit{triple}$

We define partial correctness as

$$\begin{aligned}\Gamma\models n\colon\{P\}\ t\!\succ\ \{Q\} \equiv \forall Y\ \sigma\ Z.\ \ P\ Y\ \sigma\ Z\ \longrightarrow\ \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \\ (\forall Y'\ \sigma'.\ \ \Gamma\vdash\sigma\ \xrightarrow{t:n}\ (Y',\sigma')\ \longrightarrow\ \ Q\ Y'\ \sigma'\ Z)\end{aligned}$$

Validity of $\{P\}\ t\!\succ\ \{Q\}$ intuitively means that if P holds for some type-conforming starting state σ and the evaluation of the term t terminates, then Q holds for the result and the final state σ'. Note the universal quantification on the auxiliary variable Z motivated in §5.1.2.

The predicate

$\mathsf{type\_ok} :: \mathit{prog} \to \mathit{term} \to \mathit{state} \to \mathit{bool}$
$\mathsf{type\_ok}\ \Gamma\ t\ \sigma \equiv \exists\Lambda.\ (\mathsf{normal}\ \sigma\ \longrightarrow \exists T.\ (\Gamma,\Lambda)\vdash t{::}T) \land \sigma{::}\preceq(\Gamma,\Lambda)$

expresses that the term t is well-typed (at least if σ is a normal state) and that all values in σ conform to their static types, both w.r.t. the global environment Γ and some local environment Λ. This additional precondition is required to ensure soundness, as will be discussed in §5.7.3.

5.3.2 Recursive Depth

The judgment $\Gamma \vdash \sigma \xrightarrow{t:n} (v,\sigma')$ (as well as its obvious variants for the four specific classes of terms) is a slight refinement of the evaluation judgment given in §3.2. The refinement does not alter the semantics of evaluation, *i.e.* the new parameter n is a mere annotation, stating that evaluation is done with a maximal *recursive depth* n. This notion is required for the proof of soundness and thus will be motivated in §5.7.2.

The inductive rules defining the extended judgment are exactly the same as before, except that $:n$ is added above the long arrow and the rule for unfolding the method body is replaced by

$$\frac{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{body}\ \Gamma\ C\ sig \multimap\!\succ v:n} \sigma_1}{\Gamma \vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{Methd}\ C\ sig \multimap\!\succ v:n+1} \sigma_1}$$

The original and refined version are equivalent in the sense that $\Gamma \vdash \sigma \xrightarrow{t\succ} (v,\sigma')$ iff $\exists n.\ \Gamma \vdash \sigma \xrightarrow{t:n} (v,\sigma')$, which can be shown by rule induction for each direction. The 'only if' direction relies on monotonicity:

$$\Gamma \vdash \sigma \xrightarrow{t:n} (v,\sigma') \longrightarrow \forall m.\ n \leq m \longrightarrow \Gamma \vdash \sigma \xrightarrow{t:m} (v,\sigma')$$

Note that validity is monotone in the other direction:

$$\Gamma \models n{:}\,t \longrightarrow \forall m.\ m \leq n \longrightarrow \Gamma \models m{:}\,t$$

5.3.3 Liftings

The validity of a single triple canonically carries over to sets of triples:

$prog|\models nat{:}\,(\alpha)\,triples$
$\Gamma|\models n{:}\,ts \equiv \forall t \in ts.\ \Gamma \models n{:}\,t$

More interesting is the extension of validity to assumptions, defined as

$prog,(\beta)triples|\models(\alpha)triples$
$\Gamma,A|\models ts \equiv \forall n.\ \Gamma|\models n{:}\,A \longrightarrow \Gamma|\models n{:}\,ts$

meaning that a set of triples ts is valid up to any given recursive depth under the assumption that the set A is valid up to the same depth.

Note the different type parameters for the set of triples in the assumption and in the conclusion. This emphasizes that they may have different types of auxiliary variables. Unfortunately, due to the restriction mentioned in §5.2 and the rules *asm* and *Methd* (given below) which short-circuit the type variables, assumptions and conclusions are restricted to identical types.

We abbreviate the validity of a single triple under a set of assumptions as

$$\Gamma,A \models t \equiv \Gamma,A|\models \{t\}$$

Note that our definition for $\Gamma,A|\models ts$ is weaker than the version one might expect, *viz.*

$$(\forall n.\ \Gamma|\models n{:}\,A) \longrightarrow (\forall n.\ \Gamma|\models n{:}\,ts)$$

Yet for an empty set of assumptions, both variants are equivalent, and $\Gamma,\emptyset \models t$ gives the standard notion of validity in the sense that it effectively forgets about the recursive depth:

$$\Gamma,\emptyset \models \{P\}t\succ\{Q\} \ = \ \forall n.\ \Gamma|\models n{:}\,\emptyset \longrightarrow \Gamma|\models n{:}\,\{\{P\}t\succ\{Q\}\} \ = \ \forall n.\ \forall t \in \{\{P\}t\succ\{Q\}\}.\ \Gamma \models n{:}\,t \ =$$
$$\forall Y\ \sigma\ Z\ Y'\ \sigma'.\ P\ Y\ \sigma\ Z \longrightarrow \mathsf{type\_ok}\ \Gamma\ t\ \sigma \longrightarrow \Gamma \vdash \sigma \xrightarrow{t\succ} (Y',\sigma') \longrightarrow Q\ Y'\ \sigma'\ Z$$

The *derivability judgments* have the general form

$$prog,(\beta)triples|\!\!\vdash(\alpha)triples$$

but for the standard case of a single triple in the conclusion we use the abbreviation

$$\Gamma,A\vdash t \equiv \Gamma,A|\!\!\vdash\{t\}$$

5.4 Structural Rules

As for any Hoare-style logic, there are a number of structural rules applicable for any kind of term: rules for handling assumptions and conclusions and the rule of consequence. Many logics involve further rules handling variables and logical connectives within the assertions, but with a strong rule of consequence at hand they are not really necessary.

5.4.1 Handling Conclusions

The first two rules deal with deriving finite sets of triples, which is done one by one, until finally the empty set is reached:

$$insert\ \frac{\Gamma,A\vdash t \quad \Gamma,A|\!\!\vdash ts}{\Gamma,A|\!\!\vdash\{t\}\cup ts} \qquad empty\ \frac{}{\Gamma,A|\!\!\vdash\emptyset}$$

As opposed to introducing sets of conclusions, one may throw away triples using

$$weaken\ \frac{\Gamma,A|\!\!\vdash ts' \quad ts \subseteq ts'}{\Gamma,A|\!\!\vdash ts}$$

5.4.2 Handling Assumptions

Assumptions are introduced using the *Methd* rule (*cf.* §5.6.8) and exploited using the following rule:

$$asm\ \frac{ts\subseteq A}{\Gamma,A|\!\!\vdash ts}$$

There is also a rule for throwing away assumptions:

$$(thin)\ \frac{\Gamma,A'|\!\!\vdash ts \quad A' \subseteq A}{\Gamma,A|\!\!\vdash ts}$$

It does not need to be asserted, but can be derived (with rule induction) from the others[1]. If we had given the simpler but less convenient rule

$$(asm')\ \frac{}{\Gamma,A|\!\!\vdash A}$$

for exploiting assumptions, the *thin* rule could no longer be derived.

The *cut* rule is admissible (*i.e.* valid) but not derivable:

$$(cut)\ \frac{\Gamma,A'|\!\!\vdash ts \quad \Gamma,A|\!\!\vdash A'}{\Gamma,A|\!\!\vdash ts}$$

It could be added for convenience, yet we leave it out since it is not strictly necessary for completeness.

[1] Therefore we write its name in parentheses, as we will do for all derived rules.

5.4.3 Rule of Consequence

Kleymann suggests [Sch97, §4.1 and 4.3] a rule of consequence that is stronger than the usual one because it allows to adapt the values of auxiliary variables as required. In the context of recursion this helps to avoid incompleteness and introduction of ad-hoc rules of adaptation. Hofmann [Hof97] gives a rule that is even a bit stronger. After transforming his rule to our setting, simplifying it a bit and adding result value handling, it reads as

$$(\mathit{conseq12})\ \frac{\Gamma,A\vdash\{P'\}\ t\succ\ \{Q'\} \quad \forall Y\,\sigma\ Z\ Y'\ \sigma'.\ P\ Y\,\sigma\ Z \longrightarrow (\forall Y\,Z'.\ P'\ Y\,\sigma\ Z' \longrightarrow Q'\ Y'\ \sigma'\ Z') \longrightarrow Q\ Y'\ \sigma'\ Z}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

It will turn out that for completeness we further need (derivatives of) the rule

$$(\mathit{escape})\ \frac{\forall Y\,\sigma\ Z.\ P\ Y\,\sigma\ Z \longrightarrow \Gamma,A\vdash\{\lambda Y'\ \sigma'\ Z'.\ (Y',\sigma') = (Y,\sigma)\}\ t\succ\ \{\lambda Y\,\sigma\ Z'.\ Q\ Y\,\sigma\ Z\}}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

We call it *escape* rule since it enables extrusion of the result entry, state, and auxiliary variables from the triple's precondition such that P can be used as an assumption on the meta-logical level governing the rest of the triple. This is essential in particular when dealing with dynamic binding where code depends on the state. Simple consequences of the *escape* rule are the following two:

$$(\mathit{constant})\ \frac{C \longrightarrow \Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}{\Gamma,A\vdash\{\lambda Y\,\sigma\ Z.\ C \wedge P\ Y\,\sigma\ Z\}\ t\succ\ \{Q\}}$$

$$(\mathit{impossible})\ \frac{}{\Gamma,A\vdash\{\lambda Y\,\sigma\ Z.\ \mathsf{False}\}\ t\succ\ \{Q\}}$$

As already described in [Ohe99], we noticed that rather than asserting the rules *conseq12* and *escape*, it is possible to give an even stronger rule of consequence from which these two may be derived:

$$\mathit{conseq}\ \frac{\begin{array}{l}\forall Y\,\sigma\ Z.\ P\ Y\,\sigma\ Z\ \longrightarrow (\exists P'\ Q'.\ \Gamma,A\vdash\{P'\}\ t\succ\ \{Q'\}\ \wedge\\ \qquad (\forall Y'\ \sigma'.\ (\forall Y\,Z'.\ P'\ \ Y\,\sigma\ Z' \longrightarrow Q'\ \ Y'\ \sigma'\ Z') \longrightarrow Q\ \ Y'\ \sigma'\ Z))\end{array}}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

This version is the strongest possible one since it directly reflects the semantics of the pre- and postconditions involved, as can be seen when conducting its soundness proof. Note that it allows choosing the pre- and postconditions of the inner triple, P' and Q', under the assumption P and depending on its parameters Y, σ, and Z.

Common structural rules such as

$$(\mathit{trivial})\ \frac{}{\Gamma,A\vdash\{P\}\ \ t\succ\ \{\lambda Y\,\sigma\ Z.\ \mathsf{True}\}}$$

$$(\mathit{disj})\ \frac{\Gamma,A\vdash\{P_1\}\ t\succ\ \{Q_1\} \quad \Gamma,A\vdash\{P_2\}\ t\succ\ \{Q_2\}}{\Gamma,A\vdash\{\lambda Y\,\sigma\ Z.\ P_1\ \ Y\,\sigma\ Z \vee P_2\ \ Y\,\sigma\ Z\}\ t\succ\ \{\lambda Y\,\sigma\ Z.\ Q_1\ \ Y\,\sigma\ Z \vee Q_2\ \ Y\,\sigma\ Z\}}$$

are no longer required at all and may be derived easily if desired. In contrast, two other derived rules are quite handy, namely the restriction of the rule of consequence to either the pre- or postcondition:

$$(\mathit{conseq1})\ \frac{\Gamma,A\vdash\{P'\}\ t\succ\ \{Q\} \quad P \to P'}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}} \qquad (\mathit{conseq2})\ \frac{\Gamma,A\vdash\{P\}\ t\succ\ \{Q'\} \quad Q' \to Q}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

5.5 Universal Quantification

In the rules given in the next section we will use several techniques introducing and exploiting universal quantification around triples in the rules' premises. Their common purpose is to extend the scope in which values are visible in order to reflect dependencies between different pre- and postconditions. The techniques may be nested and combined with each other: for instance, the rule for method calls (*cf.* §5.6.8) applies all of them on a single triple. We classify the techniques by the source of the values, as follows.

State Extrusion is of the form $\forall z.\ \Gamma{,}A \vdash \{P\ z\ \wedge.\ (\lambda\sigma.\ z = f\ \sigma)\ \}\ t\ z\succ\ \{Q\ z\}$, which means picking some part of the state (using some function f) in the precondition, binding it to a variable z, and using it anywhere within the triple. This technique is useful *e.g.* to save, modify locally to t, and restore local variables. It is very similar to a technique within the MGF approach that will be explained in §5.8.1, the difference being that there implicit quantification of auxiliary variables is used.

Result Extrusion, already introduced in §5.1.3, has the general form $\forall z.\ \Gamma{,}A \vdash \{P\ z {\leftarrow} \mathit{Inj}\ z\}\ t\ z\succ\ \{Q\ z\}$ where Inj is one of the injections Val, Var or Vals. It means matching (and binding) the result of the previous triple and using it anywhere within the triple, typically in order to calculate a new result or let the term t depend on it.

Value Passing involves two triples $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t_1\ z\succ\ \{Q\ z\}$ and $\forall z.\ \Gamma{,}A \vdash \{f\ (Q\ z)\ z\}\ t_2\ z\succ\ \{R\ z\}$ where f is an assertion transformer modifying $Q\ z$. It is logically equivalent to $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t_1\ z\succ\ \{Q\ z\}\ \wedge\ \Gamma{,}A \vdash \{f\ (Q\ z)\ z\}\ t_2\ z\succ\ \{R\ z\}$ and thus simply extends the scope of z to the second triple. This is useful for passing on previously obtained values to other triples.

An interesting issue is how the bound variables are actually used because this affects the usability and completeness of the resulting Hoare logic. There are again three cases:

State Transformation uses the bound variables to change the state within a pre- or postcondition, typically by applying the assertion transformer $\_\ .;\ \_$.

Result Generation uses the values to compute results to be entered in the result component of a postcondition, typically by applying the assertion transformer $\_ \leftarrow \_$.

Term Dependence means, syntactically speaking, that the term between the pre- and postcondition is an expression actually referring to the bound variables.

The first two uses are harmless: in applications one has to derive triples of the form $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t\succ\ \{Q\ z\}$ (where t does not depend on z) for all potential values of z. Concerning the Hoare logic rules, this can be done in a uniform way for some fixed but arbitrary value z because the shape of the triple is the same for all z. Only the proofs of side conditions that possibly emerge when using the rule of consequence might require enhanced proof principles like a (local) induction over z. Yet for these proofs we can assume the full power of predicate logic anyway since we consider relative completeness, as motivated in §5.1.1.

The third use, dependent terms, is problematic because the triples involved have the form $\forall z.\ \Gamma{,}A \vdash \{P\ z\}\ t\ z\succ\ \{Q\ z\}$ where the term $t\ z$ does depend on z such that uniform rule application is not possible. The only other option for a finitary proof within Hoare logic is to explicitly enumerate all possible cases for z, where of course the variety has to be finite. Often this is easy because the type of z is *bool* (or some other finite type). The other option is to derive from the precondition $P\ z$, typically by applying the *escape* rule, that only finitely many values for z — commonly even just one — are actually possible. Then the proof proceeds by constructing a finite set (or superset) of the possible values and derive the triple for each of its members where the term expression $t\ z$, as well as the assertions $P\ z$ and $Q\ z$, can be reduced to something not mentioning the variable z anymore.

5.6 Java-specific Rules

This section contains the main part of our axiomatic semantics, namely the Hoare-style rules for each kind of Java term. The rules given here describe exactly the same behavior as the operational rules of §3.2 and re-use auxiliary definitions as far as possible. Thus we do not need to repeat the general descriptions already given along with the operational semantics but concentrate on the new aspects required by the axiomatic semantics.

We have designed each rule (except for *Loop*) such that its final postcondition is given by a predicate variable only. Thus application of the rules in the typical "backward-proof" style of Hoare logic is simplified because we avoid the need for applying the rule of consequence in order to adapt the syntactical form of the postcondition, which normally requires awkward explicit instantiations. In other words, the *weakest precondition* of a given postcondition is typically generated automatically.

5.6.1 Exception Propagation

In order to describe the propagation of exceptions we use essentially the same mechanism as for the operational semantics: there is a general rule

$$Xcpt\ \frac{}{\Gamma{,}A{\vdash}\{P{\leftarrow}(\mathsf{arbitrary3}\ t)\ \wedge.\ \mathsf{Not}\circ\mathsf{normal}\}\ t{\succ}\ \{P\}}$$

stating that if an exception has occurred, the current command is ignored and thus the state is not changed. Just a suitable dummy result is generated. All other rules — except for the *Loop* rule — assume that the initial state is normal using the predicate transformer Normal.

Note that we deal with exceptions in a very simple and straightforward way by having the exception status available within the assertions, like any other part of the program state. Most other axiomatic semantics in the literature leave out exceptions completely and thus cannot infer anything in case of exceptional states, though Poetzsch-Heffter and Müller plan to extend their work [PHM99] to include them. Huisman and Jacobs [HJ00] model exceptions using special triples for exceptional states, each with its own variant of validity.

5.6.2 Standard Statements

Thanks to our implicit exception propagation mechanism, the rules for the standard statements appear almost as usual.

$$Skip\ \frac{}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ (P{\leftarrow}\bullet)\}\ .\mathsf{Skip}.\ \{P\}}$$

In order to obtain soundness w.r.t. our notion of validity, here (and in a few other rules) we have to mention explicitly the dummy result of statements, $\bullet$. In applications this is no harm since pre- and postconditions of statements do not refer to the result entry anyway.

$$Comp\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .c_1.\ \{Q\}\quad \Gamma{,}A{\vdash}\{Q\}\ .c_2.\ \{R\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .c_1\,;c_2.\ \{R\}}$$

For expression statements, the result of the expression is discarded using the $\leftarrow$ operator:

$$Expr\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ e{-}{\succ}\ \{Q{\leftarrow}\bullet\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathsf{Expr}\ e.\ \{Q\}}$$

For terms involving a condition we define the Boolean result substitution operator

$P{\leftarrow}{=}b \equiv \lambda Y\ \sigma\ Z.\ \exists v.\ (P{\leftarrow}\mathsf{Val}\ v)\ \sigma\ Z \wedge (\mathsf{normal}\ \sigma \longrightarrow \mathsf{the\_Bool}\ v = b)$

which is a variant of the operator $\leftarrow$ expressing that, unless an exception has been thrown, the result of the preceding Boolean expression is b. Using it in conjunction with the result extrusion technique introduced in §5.5 and the meta-level conditional expression $\mathsf{if}\ b\ \mathsf{then}\ \_\ \mathsf{else}\ \_$, we can describe both branches of conditional terms with a single triple, like in

$$\mathit{If}\ \frac{\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ e{-}{\succ}\ \{P'\} \qquad \forall b.\ \Gamma,A{\vdash}\{P'{\leftarrow}{=}b\}\ .(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2).\ \{Q\}}{\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2.\ \{Q\}}$$

What is a notational convenience here (to avoid two triples, one for each branch), will be essential for the *Call* rule, given below (*cf.* §5.6.8).

Another application of the Boolean substitution for case distinctions is

$$\mathit{Loop}\ \frac{\Gamma,A{\vdash}\{P\}\ e{-}{\succ}\ \{P'\} \qquad \Gamma,A{\vdash}\{\mathsf{Normal}\ (P'{\leftarrow}{=}\mathsf{True})\}\ .c.\ \{P\}}{\Gamma,A{\vdash}\{P\}\ .\mathtt{while}(e)\ c.\ \{(P'{\leftarrow}{=}\mathsf{False}){\downarrow}{=}{\bullet}\}}$$

The loop body needs to be verified only if no exception has been thrown meanwhile and the Boolean expression e yields True. Upon termination e yields False (unless an exception has occurred). Here both P and P' play the role of the loop invariant, where P' is typically equivalent to P, at least if e does not have side-effects.

Here, in order to achieve completeness, we have to give P rather than $\mathsf{Normal}\ P$ as the precondition of the triple in the rule's conclusion because it should be the same as the invariant P and in general the invariant cannot maintain the absence of exceptions. Furthermore, also for ensuring completeness, here we have to deviate from the principle of having as the final postcondition a predicate variable only. Thus applying the rule of consequence will be necessary here, but this does not cause extra nuisance since finding and inserting suitable invariants P (and P') requires manual engagement anyway.

5.6.3 Exception Handling

The rules for exception handling strongly resemble the rules given in §3.2.5, the main difference being the way intermediate results are handled.

The rule for the `throw` statement modifies the postcondition Q by updating the exception component of the state with the reference just evaluated.

$$\mathit{Throw}\ \frac{\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ e{-}{\succ}\ \{\lambda \mathsf{Val}\ a{:}.\ \mathsf{xupd}\ (\mathsf{throw}\ a)\ .;\ Q{\leftarrow}{\bullet}\}}{\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathtt{throw}\ e.\ \{Q\}}$$

When describing the effect of the statement $\mathtt{try}\ c_1\ \mathtt{catch}(C\ vn)\ c_2$ we have to distinguish whether in the state after executing c_1 an exception of appropriate (dynamic) type, *viz.* a subclass of C, is present, as denoted by $\Gamma,\sigma{\vdash}\mathsf{catch}\ C$. Only if this is the case, the statement c_2 is considered with its exception parameter vn set (using the function $\mathsf{new\_xcpt\_var}$) to the caught exception. Otherwise, the final postcondition R has to be implied immediately. For the `try ... catch` statement the $\mathsf{sxalloc}$ relation has to be lifted to the assertion level, as done by the $\mathsf{SXAlloc}$ transformer.

$$\mathit{Try}\ \frac{\begin{array}{l}\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ .c_1.\ \{\mathsf{SXAlloc}\ \Gamma\ Q\}\\ \Gamma,A{\vdash}\{Q\ \wedge.\ (\lambda\sigma.\ \ \Gamma,\sigma{\vdash}\mathsf{catch}\ C)\ ;.\ \mathsf{new\_xcpt\_var}\ vn\}\ .c_2.\ \{R\}\\ (Q\ \wedge.\ (\lambda\sigma.\ \neg\Gamma,\sigma{\vdash}\mathsf{catch}\ C)) \rightarrow R\end{array}}{\Gamma,A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathtt{try}\ c_1\ \mathtt{catch}(C\ vn)\ c_2.\ \{R\}}$$

where

$\mathsf{SXAlloc} :: \mathit{prog} \rightarrow (\alpha)\mathit{assn} \rightarrow (\alpha)\mathit{assn}$

$\mathsf{SXAlloc}\ \Gamma\ P \equiv \lambda Y\ \sigma\ Z.\ \forall\sigma'.\ \Gamma{\vdash}\sigma \xrightarrow{\mathsf{sxalloc}} \sigma' \longrightarrow P\ Y\ \sigma'\ Z$

The rule for the `finally` statement needs to transfer the exception status before executing the second substatement to the postcondition where it is combined with the current exception status, producing the final status. In earlier versions of our axiomatic semantics, we employed a special result stack entry to transfer the exception status (denoted by $\mathsf{fst}\ \sigma$ here), but this is not necessary: it suffices to apply the state extrusion technique, as explained in §5.5.

$$\mathit{Fin}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .c_1.\ \{Q\} \qquad \forall x.\ \Gamma,A\vdash\{Q\ \wedge.\ (\lambda\sigma.\ x{=}\mathsf{fst}\ \sigma)\ ;.\ \mathsf{xupd}\ (\lambda x.\ \mathsf{None})\}\ .c_2.\ \{\mathsf{xupd}\ (\mathsf{xcpt\_if}\ (x{\neq}\mathsf{None})\ x)\ .;\ R\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .c_1\ \mathtt{finally}\ c_2.\ \{R\}}$$

5.6.4 Class Initialization

In contrast to the operational rule for class initialization (*cf.* §3.2.6), here it is simpler to give two rules. If the class in question is already initialized, there is nothing to do:

$$\mathit{Done}\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{P\}}$$

Otherwise, initialization allocates a new static object, treats the superclass (if any), and finally invokes the static initializers of the class itself, whereby the current local variables are remembered in the bound variable l, hidden during the call to ini (using $\mathsf{set\_lvars\ empty}$), and later restored:

$$\mathit{Init}\ \frac{\begin{array}{c}\mathsf{the}\ (\mathsf{class}\ \Gamma\ C) = (sc,\_,\_,\_,\mathit{ini}) \qquad \mathit{sup} = \mathsf{if}\ C = \mathtt{Object}\ \mathsf{then\ Skip\ else\ init}\ sc \\ \Gamma,A\vdash\{\mathsf{Normal}\ ((P\ \wedge.\ \mathsf{Not} \circ \mathsf{initd}\ C)\ ;.\ \mathsf{supd}\ (\mathsf{init\_class\_obj}\ \Gamma\ C))\}\ .\mathit{sup}.\ \{Q\} \\ \forall l.\ \Gamma,A\vdash\{Q\ \wedge.\ (\lambda\sigma.\ l = \mathsf{locals}\ (\mathsf{snd}\ \sigma))\ ;.\ \mathsf{set\_lvars\ empty}\}\ .\mathit{ini}.\ \{\mathsf{set\_lvars}\ l\ .;\ R\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ (P\ \wedge.\ \mathsf{Not} \circ \mathsf{initd}\ C)\}\ .\mathsf{init}\ C.\ \{R\}}$$

Note that the values of sc, ini and sup depending on C are known statically and thus application of this rule simplifies immediately. See §6.3.4 for an example.

5.6.5 Simple Expressions

As already motivated in §5.1.3, the rule for literal values is

$$\mathit{Lit}\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (P{\leftarrow}\mathsf{Val}\ v)\}\ \mathsf{Lit}\ v{-}{\succ}\ \{P\}}$$

It states that for a literal expression (*i.e.* constant) v the postcondition P can be derived if P — with the value v inserted — holds as the precondition and the (pre-)state is normal. An equivalent but typically less convenient alternative form would be

$$(\mathit{Lit2})\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Lit}\ v{-}{\succ}\ \{\mathsf{Normal}\ (P{\downarrow}{=}\mathsf{Val}\ v)\}}$$

The rule for `super` is similar, except that one has to peek at the state in order to get the value of This:

$$\mathit{Super}\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (\lambda s..\ P{\leftarrow}\mathsf{Val}\ (\mathsf{val\_this}\ s))\}\ \mathtt{super}{-}{\succ}\ \{P\}}$$

Variable access is the first example where the result entry is a variable. Here we just need its first component, which is the current value of the variable.

$$\mathit{Acc}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ va{=}{\succ}\ \{\lambda\mathsf{Var}\ (v,f)\ :.\ Q{\leftarrow}\mathsf{Val}\ v\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Acc}\ va{-}{\succ}\ \{Q\}}$$

The rule for variable assignment uses the result extrusion technique to refer to the resulting of va.

$$\mathit{Ass}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ va{=}\!\!\succ\ \{Q\} \qquad \forall vf.\ \Gamma,A\vdash\{Q\leftarrow\mathsf{Var}\ vf\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ v\,{:}.\ \mathsf{assign}\ (\mathsf{snd}\ vf)\ v\ .;\ R\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ va\mathtt{:=}e{-}\!\!\succ\ \{R\}}$$

The rule for conditional expressions parallels the one for conditional statements:

$$\mathit{Cond}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_0{-}\!\!\succ\ \{P'\}\quad \forall b.\ \Gamma,A\vdash\{P'\leftarrow{=}b\}\ (\mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2){-}\!\!\succ\ \{Q\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_0\ \mathtt{?}\ e_1\ \mathtt{:}\ e_2{-}\!\!\succ\ \{Q\}}$$

The rules for type casts and the `instanceof` expression do not impose new challenges:

$$\frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ v\,{:}.\ \lambda s..\ \mathsf{xupd}\ (\mathsf{raise\_if}\ (\neg\Gamma,s\vdash v\ \mathsf{fits}\ T)\ \mathtt{ClassCast})\ .;\ Q\leftarrow\mathsf{Val}\ v\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Cast}\ T\ e{-}\!\!\succ\ \{Q\}}$$

$$\mathit{Inst}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ v\,{:}.\ \lambda s..\ Q\leftarrow\mathsf{Val}\ (\mathsf{Bool}\ (v{\neq}\mathsf{Null}\wedge\Gamma,s\vdash v\ \mathsf{fits}\ \mathsf{RefT}\ T))\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e\ \mathtt{instanceof}\ T{-}\!\!\succ\ \{Q\}}$$

5.6.6 Object Creation

Allocating an object on the heap requires lifting the halloc relation to the assertion level in analogy to the SXAlloc transformer given above. See §6.3.4 for application examples.

$$\mathit{NewC}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{\mathsf{Alloc}\ \Gamma\ (\mathsf{CInst}\ C)\ Q\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathtt{new}\ C{-}\!\!\succ\ \{Q\}}$$

where

$\mathsf{Alloc} :: \mathit{prog} \to \mathit{obj\_tag} \to (\alpha)\mathit{assn} \to (\alpha)\mathit{assn}$
$\mathsf{Alloc}\ \Gamma\ \mathit{otag}\ P \equiv \lambda Y\,\sigma\ Z.\ \forall\sigma'\ a.\ \Gamma,A\vdash\sigma\ \xrightarrow{\mathsf{halloc}\ \mathit{otag}\succ a}\ \sigma' \longrightarrow P\ (\mathsf{Val}\ (\mathsf{Addr}\ a))\ \sigma'\ Z$

$$\mathit{NewA}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init\_comp\_ty}\ T.\ \{Q\} \qquad \Gamma,A\vdash\{Q\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ i\,{:}.\ \mathsf{xupd}\ (\mathsf{check\_neg}\ i)\ .;\ \mathsf{Alloc}\ \Gamma\ (\mathsf{Arr}\ T\ (\mathsf{the\_Intg}\ i))\ R\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathtt{new}\ T[e]{-}\!\!\succ\ \{R\}}$$

5.6.7 Variables

The rule for local variables is analogous to the rule for the `super` expression:

$$\mathit{LVar}\ \frac{}{\Gamma,A\vdash\{\mathsf{Normal}\ (\lambda s..\ P\leftarrow\mathsf{Var}\ (\mathsf{lvar}\ vn\ s))\}\ \mathsf{LVar}\ vn{=}\!\!\succ\ \{P\}}$$

The rules for field and array variables have a common pattern: calling a variable-generating function vf on the state and applying a given assertion to the resulting variable and state. We capture this in the predicate transformer [Factoring]

$\_\ ..;\ \_ :: (\mathit{state} \to \mathit{vvar} \times \mathit{state}) \to (\alpha)\mathit{assn} \to (\alpha)\mathit{assn}$
$vf\ ..;\ P \equiv \lambda Y\,\sigma.\ \mathsf{let}\ (v,\sigma') = vf\ \sigma\ \mathsf{in}\ P\ (\mathsf{Var}\ v)\ \sigma'$

Applying it, we obtain the two rules

$$\mathit{FVar}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ C.\ \{Q\} \qquad \Gamma,A\vdash\{Q\}\ e{-}\!\!\succ\ \{\lambda\mathsf{Val}\ a\,{:}.\ \mathsf{fvar}\ C\ \mathit{stat}\ \mathit{fn}\ a\ ..;\ R\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \{C,\mathit{stat}\}e..\mathit{fn}{=}\!\!\succ\ \{R\}}$$

$$\mathit{AVar}\ \frac{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_1{-}\!\!\succ\ \{Q\} \quad \forall a.\ \Gamma,A\vdash\{Q\leftarrow\mathsf{Val}\ a\}\ e_2{-}\!\!\succ\ \{\lambda\mathsf{Val}\ i\,{:}.\ \mathsf{avar}\ \Gamma\ i\ a\ ..;\ R\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e_1[e_2]{=}\!\!\succ\ \{R\}}$$

5.6.8 Method Call

A rather complex issue within an axiomatic semantics in general is *mutual recursion*. For an object-oriented language, *dynamic binding* in method calls gives a further challenge.

Dynamic Binding

Handling dynamic binding for method calls is difficult for two reasons.

First, the actual method to be called depends on the class D dynamically computed from the receiver expression e and thus in general cannot be inferred statically. The usual technique for dealing with term dependence, as done *e.g.* for the standard Hoare rule for conditional statements, is to statically enumerate all possible values. We cannot use it for D because the variety of possible values is large — but finite because it is bound by the total number of methods in the given program — and not fixed locally since it depends on the class hierarchy. We handle this problem with the state extrusion and term dependence technique introduced in §5.5, introducing universal quantification for D and the precondition $(\lambda(x,s).\ D = \mathsf{target}\ mode\ s\ a\ md \wedge \ldots)$ binding D. An alternative solution is given in [PHM99], where D is referred to via This and the possibly large range of values for D is handled in a cascadic way using two special rules.

Second, one should be able to assume that for invocation mode `interface` or `virtual` the actual value D is a subtype of t, the (static) type of e. The intuitive — but absolutely non-trivial — reason why the relation $\mathsf{Class}\ D \preceq \mathsf{RefT}\ t$ holds is of course type safety. The problem here is how to establish this relation. The rules given in [PHM99], for example, put the burden of verifying the relation on the user, which is a legal option, but in general not practically feasible. In contrast, our solution makes the relation available to the user as a helpful assumption, which transfers the proof burden once and for all to the soundness proof on the meta-level.

We write the subtype relation in the form

$$prog \vdash inv\_mode \rightarrow tname \preceq ref\_ty$$

and give it the definition

$$\begin{array}{l}\Gamma \vdash mode \rightarrow D \preceq t \equiv mode = \mathsf{IntVir} \longrightarrow \\ \qquad\qquad \mathsf{is\_class}\ \Gamma\ D \wedge (\mathsf{if}\ (\exists T.\ t{=}\mathsf{ArrayT}\ T)\ \mathsf{then}\ D{=}\mathtt{Object}\ \mathsf{else}\ \Gamma \vdash \mathsf{Class}\ D \preceq \mathsf{RefT}\ t)\end{array}$$

reflecting the knowledge on D required for program verification in case of virtual method invocation.

A minor further complication is that we have to transfer the result a of the expression e not to the triple directly following but to the one after it. We can cope with this by combining the result extrusion and value passing techniques.

The remaining parts of the method call rule deals with the unproblematic issues of argument evaluation, setting up the local variables (including parameters) of the called method and restoring the previous local variables on return, for which we use the universally quantified variable l. See §6.3.4 for an application example.

$$Call\ \frac{\begin{array}{c}\Gamma,A \vdash \{\mathsf{Normal}\ P\}\ e{-}\!\succ\ \{Q\} \qquad \forall a.\ \Gamma,A \vdash \{Q {\leftarrow} \mathsf{Val}\ a\}\ args \doteq\!\succ\ \{R\ a\} \\ \forall a\ vs\ D\ l.\ \Gamma,A \vdash \{(R\ a {\leftarrow} \mathsf{Vals}\ vs\ \ \wedge.\ (\lambda(x,s).\ D = \mathsf{target}\ mode\ s\ a\ md \wedge l = \mathsf{locals}\ s)\ ;. \\ \mathsf{init\_lvars}\ \Gamma\ D\ (mn,pTs)\ mode\ a\ vs) \wedge.\ (\lambda\sigma.\ \mathsf{normal}\ \sigma \longrightarrow \Gamma \vdash mode \rightarrow D \preceq t)\} \\ \mathsf{Methd}\ D\ (mn,pTs){-}\!\succ\ \{\mathsf{set\_lvars}\ l\ .;\ S\}\end{array}}{\Gamma,A \vdash \{\mathsf{Normal}\ P\}\ \{t,md,mode\}e..mn(\{pTs\}args){-}\!\succ\ \{S\}}$$

Note that $\Gamma \vdash mode \rightarrow D \preceq t$ is asserted only if the current state is normal. Otherwise, an exception occurred evaluating e (or $args$) such that we cannot assume anything about D.

For the same reason the well-typedness judgment in the definition of $\mathsf{type\_ok}$ (*cf.* §5.3.1) is guarded by $\mathsf{normal}\ \sigma$ since otherwise $\mathsf{Methd}\ D\ (mn,pTs)$ is not necessarily well-typed.

The above rule is general enough to handle all sorts of method calls, also static ones and those relative to interfaces or `super`. One may derive simpler specialized versions of the *Call* rule, for example for static method calls:

$$(CallS)\ \frac{\begin{array}{c}\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ e\!-\!\succ\{Q\}\quad \Gamma,A\vdash\{Q\!\downarrow\}\ args\doteq\!\succ\{R\}\\ \forall vs\ l.\ \Gamma,A\vdash\{R\!\leftarrow\!\mathsf{Vals}\ vs\ \wedge.\ (\lambda\sigma.\ l=\mathsf{locals}\ (\mathsf{snd}\ \sigma))\ ;.\ \mathsf{init\_lvars}\ \Gamma\ C\ (mn,pTs)\ \mathsf{Static}\ a\ vs\}\\ \mathsf{Methd}\ C\ (mn,pTs)\!-\!\succ\{\mathsf{set\_lvars}\ l\ .;\ S\}\end{array}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \{t,\mathsf{ClassT}\ C,\mathsf{Static}\}e..mn(\{pTs\}args)\!-\!\succ\{S\}}$$

Note that the rule can ignore (applying $\downarrow$) the value of e, and the free variable a acts as a dummy parameter of the function $\mathsf{init\_lvars}$.

From the methodological perspective it would be interesting to take into account semantic subtyping. Doing so, one could introduce method specifications, in particular within interfaces, that have to be fulfilled by all classes implementing them (and their subclasses). Then a derived rule for method calls could exploit the semantic subtyping in the sense that for verifying calls to methods bound by a specification one only has to show that the method specification, as statically determined by the type annotation t, fulfils the given requirements (rather than directly showing them for all method implementations in subclasses of t).

Mutual Recursion

We cope with recursive calls adopting the standard solution of introducing Hoare triples as assumptions within the derivation judgments. Thanks to the well-known recursion rule (see *e.g.* [Apt81, §3.2] and [PHM99]), within an unfolded method body one may appeal to a suitable assumption catering for all further recursive calls.

As discussed in [Ohe99], the recursion rule is sufficient for completeness, but its nested application required for mutual recursion in general gives rise to replication of proofs for part of the methods involved. This nuisance can be overcome with rules that allow *simultaneous* rather than nested verification. One such rule is given by Homeier and Martin in [HM96]. Since they aim at verification condition generation, they designed a rule for verifying all procedures contained in a program simultaneously, which requires the user to identify in advance a single specification for each method suitable to cover all invocation contexts. Our rule can also be used to verify all methods at once, but is more flexible for interactive verification: each time a call to a cluster of mutually recursive procedures is encountered, it allows verifying simultaneously as many (and no more) procedures as desired and to identify the necessary specifications locally.

The *Methd* rule allows verifying the specifications of a set of methods ms (or to be exact, method implementations) by verifying their expansions, that is, the corresponding method bodies. It is vital that when encountering recursive calls during the verification of the bodies, one can assume that the method implementations already fulfill their specifications. This explains why method implementations are separated from method bodies, something that would not be necessary for the underlying operational semantics itself.

$$Methd\ \frac{\Gamma,A\cup\{\{P\}\ \mathsf{Methd}\!-\!\succ\{Q\}\mid ms\}\ \Vdash\ \{\{P\}\ \mathsf{body}\ \Gamma\!-\!\succ\{Q\}\mid ms\}}{\Gamma,A\Vdash\{\{P\}\ \mathsf{Methd}\!-\!\succ\{Q\}\mid ms\}}$$

where $\{\{P\}\ tf\!-\!\succ\{Q\}\mid ms\}\equiv(\lambda(C,sig).\ \{\mathsf{Normal}\ (P\ C\ sig)\}\ tf\ C\ sig\!-\!\succ\{Q\ C\ sig\})\ \text{``}\ ms$ yields a family of method triples indexed by the set ms (consisting of pairs of a class and signature). Both the assertions P and Q and the term function tf depend on the index values given by ms, such that members like $\{\mathsf{Normal}\ (P\ C\ sig)\}\ \mathsf{Methd}\ C\ sig{-}\!\succ\ \{Q\ C\ sig\}$ are generated.

The structural rules for handling assumptions and sets of triples in the conclusion have been described in §5.4.

The function body (*cf.* §3.2.9) and the constructor Body for intermediate body terms have been introduced because the *Methd* rule above is already complex enough. After the function body Γ has obtained a class name and a signature during applications of the *Methd* rule, handling the method body is the same as in the operational semantics: the expression body $\Gamma\ C\ sig$ calculates the intermediate term Body $D\ c\ e$ with the entries for the defining class of the method, the actual method body, and the result expression. These are then processed sequentially using the rule

$$\mathit{Body}\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ .\mathsf{init}\ D.\ \{Q\}\qquad \Gamma{,}A{\vdash}\{Q\}\ .c.\ \{R\}\qquad \Gamma{,}A{\vdash}\{R\}\ e{-}\!\succ\ \{S\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ \mathsf{Body}\ D\ c\ e{-}\!\succ\ \{S\}}$$

5.6.9 Expression Lists

Lists of expressions are dealt with canonically:

$$\mathit{Nil}\ \frac{}{\{\mathsf{Normal}\ P{\leftarrow}\mathsf{Vals}\ []\}\ []\dot{=}\!\succ\ \{P\}}$$

$$\mathit{Cons}\ \frac{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ e{-}\!\succ\ \{Q\}\qquad \forall v.\ \Gamma{,}A{\vdash}\{Q{\leftarrow}\mathsf{Val}\ v\}\ es\dot{=}\!\succ\ \{\lambda \mathsf{Vals}\ vs :.\ R{\leftarrow}\mathsf{Vals}\ (v\#vs)\}}{\Gamma{,}A{\vdash}\{\mathsf{Normal}\ P\}\ e\#es\dot{=}\!\succ\ \{R\}}$$

5.6.10 Critical Review

The inductively defined Hoare logic rules given in this section precisely cover the axiomatic semantics of Java$^{\ell ight}$. Unfortunately, they are not easy to read and to apply by hand. The reason for this is the inherent complexity of the language, requiring in particular non-trivial transformations of the state. On the other hand, the format of our rules should facilitate the construction of an automatic verification condition generator, and our experience with example proofs (§6.3.4) shows that even for rather complex assertions the theorem proving system deals with proof obligations mostly automatically.

One might also get the impression that our axiomatic semantics is rather close to the operational semantics. We believe that this similarity is not intrinsic but due to our use of the same state transformers as for the operational semantics. We do this for simplicity in conjunction with our semantical notion of assertions, but it should be possible to give a more syntactic notion of assertions and corresponding rules that do not refer to any concept of the operational semantics. The general advantages of a Hoare logic over the operational semantics for program verification, namely concentration on the actually relevant properties of the state and powerful tools for dealing with loops and recursion, are fully realized by our axiomatic semantics.

5.7 Soundness

A Hoare logic that is unsound would be useless since its very purpose is to verify correctness of programs. Thus after giving a Hoare logic the proof of its soundness is obligatory, in particular when — like in our case — the rules are rather involved and thus their correctness is by far not obvious.

5.7.1 General Approach

The ultimate goal for proving soundness of our axiomatic semantics w.r.t. the operational semantics is

$$\mathsf{wf\_prog}\ \Gamma \ \longrightarrow\ \Gamma,\emptyset\vdash t \ \longrightarrow\ \Gamma,\emptyset\models t$$

i.e. any triple t that is derivable from the empty set of assumptions is valid. The additional premise that the program is well-formed is required to show soundness of the method call rule requiring type safety, as explained in §5.6.8 and 5.7.3.

The soundness goal is a direct instance of

$$\mathsf{wf\_prog}\ \Gamma \ \longrightarrow\ \Gamma,A\|\!\!\vdash ts \ \longrightarrow\ \Gamma,A\|\!\!\models ts$$

which can be shown as usual by rule induction on the derivation of $\|\!\!\vdash$.

The different cases emerging in the induction are basically straightforward, with a few notable exceptions. Since the *Loop* rule involves a loop invariant rather than unfolding the loop as done for the operational semantics, it requires an auxiliary rule induction on the derivation of the evaluation judgment as contained in the definition of validity.

5.7.2 Method Implementation Rule

The *Methd* rule demands special treatment because it adds assumptions about recursive calls, such that an (inductive) argument on the depth of these calls is needed in order to avoid circularities. This could be achieved by syntactic manipulations that unfold procedure calls up to a given depth n, as done *e.g.* in [Hof97]. Instead, we prefer a semantic approach inspired by the proofs given in [PHM99] and [Kle98]: employing the notion of recursive depth already introduced in §5.3.2.

Induction on the recursive depth boils down to showing in the base case

$$\Gamma\models 0{:}\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ C\ sig{-}\!\!\succ\ \{Q\}$$

and in the inductive step

$$\Gamma\models n{:}\{\mathsf{Normal}\ P\}\ \mathsf{body}\ \Gamma\ C\ sig{-}\!\!\succ\{Q\} \ \longrightarrow\ \Gamma\models n{+}1{:}\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ C\ sig{-}\!\!\succ\ \{Q\}$$

5.7.3 Method Call Rule and Type safety

The *Call* rule is not only bulky (and thus it helps to treat this case separately from the main rule induction) but also raises major semantical complications. Interestingly, type safety plays a crucial role here: The important fact that for virtual method calls the relation $\Gamma\vdash mode{\rightarrow}D{\preceq}rt$ holds, can be derived in general only if the method call is well-typed and the state in which the class D has been dynamically looked up conforms to its environment.

In order to obtain the desired conformance property, one essentially has to keep it as an invariant. But rather than requiring the user to prove this property over and over for each program to be verified, we built it — together with well-typedness — into our notion of validity (*cf.* §5.3.1) in the form of the judgment $\mathsf{type\_ok}$. This also gives rise to a new rule:

$$hazard\ \frac{}{\Gamma,A\vdash\{P\ \wedge.\ \mathsf{Not}\circ\mathsf{type\_ok}\ \Gamma\ t\}\ t\!\succ\ \{Q\}}$$

The rule, which will be required for the completeness proof, indicates that if at any time conformance was violated, anything could happen — something that is in line with our intuition on erroneous program execution.

Including conformance (and well-typedness) into validity complicates the proof of soundness, because now we have to show that it is an invariant property of any valid triple, affecting each case of the main rule induction, not only the one for method calls. Fortunately, we have already proved type soundness for the operational semantics, as given in §4. Making use of this theorem can be simplified — yet not hidden completely — by using for the main induction actually a variant of our validity notion, namely

$$\begin{aligned}\Gamma{\models}n{:}\{P\}\ t{\succ}\ \{Q\} \equiv \forall Y\,\sigma\ Z.\ \ P\ Y\,\sigma\ Z\ \longrightarrow\ \forall\Lambda\ T.\ \sigma{::}{\preceq}(\Gamma,\Lambda) \longrightarrow (\mathsf{normal}\ \sigma \longrightarrow (\Gamma,\Lambda){\vdash}t{::}T) \\ \longrightarrow \forall Y'\ \sigma'.\ \Gamma{\vdash}\sigma \xrightarrow{t:n} (Y',\sigma')\ \longrightarrow\ Q\ Y'\ \sigma'\ Z \wedge \sigma'{::}{\preceq}(\Gamma,\Lambda)\end{aligned}$$

One can show easily, exploiting type soundness, that for well-formed programs this variant is equivalent to the original one. Even with this variant, parts of the type soundness proof have to be repeated, *e.g.* to derive well-typedness of the static initializer invoked for class initialization.

5.7.4 Summary

We have proven

Theorem 2 *For well-formed programs our axiomatic semantics is sound w.r.t. its operational counterpart.*

As we can conclude from this section, one interesting aspect of the proof of soundness is to find a suitable notion of validity capable of capturing an inductive argument on the recursive depth of procedure calls. The other noticeable thing is the insight that type safety is required to prove correct the rule for handling dynamic binding.

5.8 Completeness

The proof of completeness, stating that the given Hoare logic is useful (at least from the theoretical perspective), is much more challenging than the proof of soundness. We give the outline of the proof in a bit more detail since it is the first such proof for an object-oriented language.

We benefit heavily from the MGF approach which is described below. We extend this approach, which was given for only a single recursive procedure, to mutually recursive methods and static initialization using auxiliary inductions. As discussed in [Ohe99], when dealing with mutual recursion some complications arise, which could be overcome in three different ways, each with specific advantages and drawbacks. Here we implement the first two variants involving structural induction that either is nested as deep as the number of methods involved or handles all these methods simultaneously. The third variant, not used here, employs rule induction on the operational semantics, which is more powerful and would save a lot of effort avoiding the auxiliary inductions. On the other hand, it requires an unpleasant unfolding variant of the *Loop* rule and an additional divergence rule. Thus it is probably too tightly connected to the operational semantics, and the employment of rule inductions makes its usability at least doubtful in the light of the proof-theoretical remarks given in §5.8.5.

Our ultimate goal for proving (relative) completeness is to show that for a well-structured program, any valid triple is derivable from the empty set of assumptions:

$$\mathsf{ws\_prog}\ \Gamma\ \longrightarrow\ \Gamma,\emptyset{\models}t\ \longrightarrow\ \Gamma,\emptyset{\vdash}t$$

The well-known approach involving weakest preconditions $\mathsf{wp}\ t\ Q$ for a given term t and postcondition Q cannot be pursued here, because when verifying recursive method calls the postcondition changes such that structural induction on t does not go through.

5.8.1 MGF Approach

The *Most General Formula (MGF)* approach was introduced by Gorelick [Gor75] and promoted by Apt [Apt81], Kleymann [Kle98] and others.

For partial correctness, the MGF of a term t gives for the most general precondition (which just remembers the initial state) the strongest postcondition, which is the operational semantics of t. More precisely, the MGF of t in the context of a program Γ is defined as

$\{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}$
where
$\{P\}\ t\succ\ \{\Gamma\rightarrow\} \equiv \{P\}\ t\succ\ \{\lambda Y\,\sigma'\ \sigma.\ \Gamma\vdash\sigma\ \xrightarrow{t\succ}\ (Y,\sigma')\}$
$\doteq\ \equiv \lambda Y\,\sigma\ Z.\ \sigma = Z$

Note that here the auxiliary variables have the type *state* since they refer to the initial program state. In the precondition the state is stored in Z and retrieved in the postcondition in the form of the bound variable σ. Using it, the postcondition asserts that the result Y and state σ' are exactly those obtained from the initial state by evaluating t. Thus the MGF is trivially valid, and the main task is to show that it is also derivable.

A property of the MGF used often is that $\Gamma,A\vdash\{\mathsf{Normal}\ \doteq\}\ t\succ\ \{\Gamma\rightarrow\}$ can be interchanged freely with $\Gamma,A\vdash\{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}$, *i.e.* restricting the precondition to normal states is sufficient, because the case of an exceptional state can be always dealt with the *Xcpt* rule.

The main lemma of the MGF approach is

$$\mathit{MGF\_deriv}\quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \Gamma,\emptyset\vdash\{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}$$

Once the derivability of the MGF has been proved, completeness is a rather simple consequence, as follows. We show

$\Gamma,\emptyset\models\{P\}\ t\succ\ \{Q\} \longrightarrow\ \Gamma,\emptyset\vdash\{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}\ \longrightarrow\ \Gamma,\emptyset\vdash\{P\}\ t\succ\ \{Q\}$

First, we apply the rule

$$(\mathit{no\_hazard})\ \frac{\Gamma,A\vdash\{P\ \wedge.\ \mathsf{type\_ok}\ \Gamma\ t\}\ t\succ\ \{Q\}}{\Gamma,A\vdash\{P\}\ t\succ\ \{Q\}}$$

derived from *hazard* (*cf.* §5.7.3), in order to obtain the extra precondition $\mathsf{type\_ok}\ \Gamma\ t$ which is needed because this judgment is part of our notion of validity. Next, we apply the *conseq12* rule, and after unfolding the definitions of validity we are left with the proof obligation

$(\forall n\ Y\,\sigma\ Z.\ (\forall t{\in}\emptyset.\ \Gamma\models n{:}\,t)\ \longrightarrow\ P\ Y\,\sigma\ Z\ \longrightarrow\ \mathsf{type\_ok}\ \Gamma\ t\ \sigma\ \longrightarrow$
$\quad(\forall Y'\ \sigma'.\ \Gamma\vdash\sigma\ \xrightarrow{t:n}\ (Y',\sigma')\ \longrightarrow\ Q\ Y'\ \sigma'\ Z))\ \longrightarrow$
$\forall Y\,\sigma\ Z.\ P\ Y\,\sigma\ Z \wedge \mathsf{type\_ok}\ \Gamma\ t\ \sigma\ \longrightarrow\ (\forall Y'\ \sigma'.\ \Gamma\vdash\sigma\ \xrightarrow{t\succ}\ (Y',\sigma') \longrightarrow Q\ Y'\ \sigma'\ Z)$

which is a rather trivial predicate-logical theorem, exploiting the fact

$\Gamma\vdash\sigma\ \xrightarrow{t\succ}\ (w,\sigma')\ \ \longrightarrow\ \exists n.\ \Gamma\vdash\sigma\ \xrightarrow{t:n}\ (w,\sigma')$

The main lemma is proved basically by structural induction. Complications arise because for method calls as well as class initialization the terms involved do not become structurally smaller. To solve the problem we employ auxiliary inductions on the number of methods not yet verified and on the number of classes not yet initialized, as explained in the two following subsections.

5.8.2 Mutual Recursion

The idea for handling mutual recursion is as follows. First prove derivability of the MGF under the assumption that it has already been proved for all methods:

$$\textit{MGF\_asm} \quad \begin{array}{l}(\forall C\ sig.\ \mathsf{is\_methd}\ \Gamma\ C\ sig \longrightarrow \Gamma,A\vdash\{\doteq\}\ \mathsf{In1l}\ (\mathsf{Methd}\ C\ sig)\succ\ \{\Gamma\!\rightarrow\}) \longrightarrow \\ \Gamma,A\vdash\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}\end{array}$$

Then prove *MGF_deriv* applying the lemma, the *Methd* rule, which supplies the required assumptions for the methods, and the *asm* rule for exploiting them. There are two alternatives for collecting the assumptions. Both alternatives rely on the fact that for a well-structured program the number of methods to consider is finite:

$$\textit{finite\_is\_methd} \quad \mathsf{ws\_prog}\ \Gamma \longrightarrow \mathsf{finite}\ \{(C,sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$$

It is interesting to note that for the whole proof of completeness — in contrast to soundness — well-formedness is not required at all, and the only occasion where we need well-structuredness is to ensure finiteness.

Nested Version

One alternative is to use the classical recursion rule that adds just one assumption per application:

$$(\textit{MethdN})\ \frac{\Gamma,A \cup \{\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ \ C\ sig\!\rightarrow\!\succ\ \{Q\}\}\vdash\ \{\mathsf{Normal}\ P\}\ \mathsf{body}\ \Gamma\ C\ sig\!\rightarrow\!\succ\ \{Q\}}{\Gamma,A\vdash\{\mathsf{Normal}\ P\}\ \mathsf{Methd}\ C\ sig\!\rightarrow\!\succ\ \{Q\}}$$

A minor advantage of this version is that it does not require the rules *empty* and *insert* for handling sets of conclusions. The main disadvantage is that it requires a complicated scheme for induction on the number of methods not yet considered:

$$\frac{\mathsf{finite}\ U \qquad uA = (\lambda(C,\ sig).\ \{\doteq\}\ \mathsf{In1l}\ (\mathsf{Methd}\ C\ sig)\succ\ \{\Gamma\!\rightarrow\})\ ``\ U}{\forall A.\ A \subseteq uA \longrightarrow n \leq |uA| \longrightarrow |A| = |uA| - n \longrightarrow \forall t.\ \Gamma,A\vdash\{\doteq\}\ t\succ\ \{\Gamma\!\rightarrow\}}$$

which is proved by induction on n. It is applied instantiating U to $\{(C,sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$, n to $|U|$, and consequently A to $\emptyset$, yielding the desired result. Note that without finiteness, calculations on cardinality like $|A| = |uA| - n$ would be meaningless.

The induction scheme has been inspired by Hofmann [HO99]. His lecture notes [Hof97] further contain a proof of completeness using the MGF approach for the language IMP augmented by a single procedure, which we have simplified and extended for our purposes.

Simultaneous Version

We invented the *Methd* rule that allows handling procedures simultaneously not only in order to simplify applications, but also to make the complicated nesting scheme of the first version dispensable for the meta-theoretic completeness proof. Using its power, the second version becomes rather straightforward. In this case, the *cut* rule would be convenient, but it can be circumvented by employing the derived rule

$$\frac{F \subseteq U \qquad \mathsf{finite}\ U \qquad ((\forall(C,sig){\in}F.\ \Gamma,A\vdash f\ C\ sig) \longrightarrow (\forall(C,sig){\in}U.\ \Gamma,A\vdash g\ C\ sig))}{\Gamma,A|\!\vdash(\lambda(C,sig).\ f\ C\ sig)\ ``\ F \longrightarrow \Gamma,A|\!\vdash(\lambda(C,sig).\ g\ C\ sig)\ ``\ F}$$

which is proved by induction on the size of U. On application, both F and U get instantiated to $\{(C,sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$, so finiteness of the number of methods is vital also here.

5.8.3 Static Initialization

Now it remains to show $\Gamma,A \vdash \{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}$ under the assumption that the MGFs for all proper methods are derivable:

$$\forall C\ sig.\ \mathsf{is\_methd}\ \Gamma\ C\ sig \longrightarrow \Gamma,A \vdash \{\doteq\}\ \mathsf{In1l}\ (\mathsf{Methd}\ C\ sig)\succ\ \{\Gamma\rightarrow\}$$

The precondition of the assumption can be discharged because due to the relation $\mathsf{type\_ok}$ in our notion of validity, we get the fact that $\mathsf{Methd}\ C\ sig$ is well-typed for free. Furthermore, all well-typed methods are proper, as follows easily from the corresponding definitions:

$$\textit{wt\_Methd\_is\_methd} \quad (\Gamma,\Lambda)\vdash \mathsf{In1l}\ (\mathsf{Methd}\ C\ sig)::T \longrightarrow \mathsf{is\_methd}\ \Gamma\ C\ sig$$

Static initialization requires an induction on the number of classes not yet initialized. To this end we define the auxiliary concepts

$$\begin{aligned}
&\mathsf{nyinitcls} :: \textit{prog} \rightarrow \textit{state} \rightarrow (\textit{tname})\textit{set}\\
&\mathsf{nyinitcls}\ \Gamma\ \sigma \equiv \{C.\ \mathsf{is_class}\ \Gamma\ C \wedge \neg\ \mathsf{initd}\ C\ \sigma\}\\[1ex]
&_\vdash\mathsf{init}\leq_ :: \textit{prog} \rightarrow \textit{nat} \rightarrow \textit{state} \rightarrow \textit{bool}\\
&\Gamma\vdash\mathsf{init}\leq n \equiv \lambda\sigma.\ |\mathsf{nyinitcls}\ \Gamma\ \sigma| \leq n\\[1ex]
&\{=:_\}\ _\succ\ \{_\rightarrow\} :: \textit{nat} \rightarrow \textit{term} \rightarrow \textit{prog} \rightarrow (\textit{state})\textit{triple}\\
&\{=:n\}\ t\succ\ \{\Gamma\rightarrow\} \equiv \{\doteq \wedge.\ \Gamma\vdash\mathsf{init}\leq n\}\ t\succ\ \{\Gamma\rightarrow\}
\end{aligned}$$

such that $\mathsf{nyinitcls}\ \Gamma\ \sigma$ is the set of classes not yet initialized, or to be more precise, whose initialization has not yet begun, in state σ. The triple $\{=:n\}\ t\succ\ \{\Gamma\rightarrow\}$ is a variant of the MGF with the extra precondition that the number of classes not yet initialized is not greater than n.

$\mathsf{nyinitcls}\ \Gamma\ \sigma$ is finite because it is a subset of the finite set of proper classes. It cannot grow (and thus increase its cardinality) during program execution, as captured by the lemma

$$\textit{nyinitcls\_gext} \quad \mathsf{snd}\ \sigma \trianglelefteq \mathsf{snd}\ \sigma' \longrightarrow \mathsf{nyinitcls}\ \Gamma\ \sigma' \subseteq \mathsf{nyinitcls}\ \Gamma\ \sigma$$

and it actually shrinks by one when a class is newly initialized.

Since $\Gamma,A \vdash \{\doteq\}\ t\succ\ \{\Gamma\rightarrow\}$ is equivalent to $\forall n.\ \Gamma,A \vdash \{=:n\}\ t\succ\ \{\Gamma\rightarrow\}$, it remains to show

$$\forall n\ C\ sig.\ \Gamma,A \vdash \{=:n\}\ \mathsf{In1l}(\mathsf{Methd}\ C\ sig)\succ\ \{\Gamma\rightarrow\} \longrightarrow \forall t.\ \Gamma,A \vdash \{=:n\} t\succ \{\Gamma\rightarrow\}$$

We can do this by full induction on n, *i.e.* we may assume that $\forall t.\ \Gamma,A \vdash \{=:m\}\ t\succ\ \{\Gamma\rightarrow\}$ already holds for all smaller m.

5.8.4 Main Induction

Finally, we have collected enough assumptions such that the main induction will go through. We show

$$\begin{aligned}
&(\forall n\ C\ sig.\ \Gamma,A \vdash \{=:n\}\ \mathsf{In1l}\ (\mathsf{Methd}\ C\ sig)\succ\ \{\Gamma\rightarrow\}) \longrightarrow\\
&(\forall m.\ m<n \longrightarrow (\forall t.\ \Gamma,A \vdash \{=:m\}\ t\succ\ \{\Gamma\rightarrow\})) \longrightarrow\\
&\quad \Gamma,A \vdash \{=:n\}\ t\succ\ \{\Gamma\rightarrow\}
\end{aligned}$$

by structural induction on t. We comment on the most interesting cases.

The first premise is exploited for handling the Methd expression itself and its application in the *Call* case simply by assumption.

We prove the case of the initialization statement as the separate lemma

$$\mathit{MGFn\_init} \quad (\forall m.\ m{<}n \longrightarrow \forall t.\ \Gamma{,}A{\vdash}\{{=}{:}\,m\}\ t{\succ}\ \{\Gamma{\rightarrow}\}) \longrightarrow \Gamma{,}A{\vdash}\{{=}{:}\,n\}\ \mathsf{In1r}\ (\mathsf{init}\ C){\succ}\ \{\Gamma{\rightarrow}\}$$

because it is needed several times, namely for all terms that involve potential class initialization. When applying the lemma, we can of course make use of the second premise. The actual use of the premise is in the proof of *MGFn_init* itself: for handling the initialization of the superclass and the static initializer of the current class where we know that the current class is actually being initialized and thus the number of classes not yet initialized decreases. This proof is one of the rare cases where we take advantage of the implicit precondition that the current term (init C here) is well-typed and thus C is a proper class.

For the *Loop* case, we employ an alternative formulation of the MGF, augmented by a predicate p in the precondition:

$$\Gamma{,}A{\vdash}\{\mathsf{Normal}\ (\doteq \wedge.\ p)\}\ t{\succ}\ \{\Gamma{\rightarrow}\}\ =$$
$$\Gamma{,}A{\vdash}\{\mathsf{Normal}\ ((\lambda Y\,\sigma\ Z.\ \forall w\sigma'.\ \Gamma{\vdash}\sigma \xrightarrow{t\succ} w\sigma' \longrightarrow w\sigma' = Z) \wedge.\ p)\}\ t{\succ}\ \{\lambda Y\,\sigma\ Z.\ (Y{,}\sigma) = Z\}$$

In the second line the auxiliary variable Z stores the result and final — rather than initial — state, which allows formulating a suitable loop invariant. We make use only of the "if" direction of the above equivalence, which is also the more interesting one: its proof relies on the facts that evaluation is deterministic and that there are at least two different program states.

Note that the alternative version of the MGF has a different type of auxiliary variable, namely $\mathit{res} \times \mathit{state}$. Therefore, we have to apply the *conseq* and *Loop* rules with rather general types. Yet due to the type restrictions of HOL mentioned in §5.2, the original rules from the inductive definition are not general enough, and thus we have to state variants of them with identical propositions but generalized types as axioms.

5.8.5 Proof-theoretical Remarks

One might wonder if the implication proved,

$$\mathsf{ws\_prog}\ \Gamma \ \longrightarrow\ \Gamma{,}\emptyset{\models}t \ \longrightarrow\ \Gamma{,}\emptyset{\vdash}\,t$$

really means relative completeness, in the sense that for any given concrete program Γ and valid triple t there is a derivation for t, requiring only finitely many applications of the Hoare logic rules plus a complete predicate logic for dealing with side conditions like in the rule of consequence.

One potential problem is the fact that the rules *Methd* and *Init* yield structural expansion of terms rather than reduction of subterms, as already noted and dealt with in the previous subsections.

Furthermore, it is a general property of inductively defined sets S that any proof tree for a theorem $x \in S$ has a finite height but possibly infinite width, for example if the definition of S contains a rule

$$\frac{\forall y.\ f\ y{\in}S}{z{\in}S}$$

where y is of an infinite type. Since in particular for our inductively defined relation $\_{,}\_{\|}{\vdash}\_$ there are rules involving universal quantifications on possibly infinite domains, for example parts of the program state, we cannot be sure a priori that any proof of $\Gamma{,}A{\|}{\vdash}\,ts$ is finitary.

Thus we have to ensure ourselves of finiteness by other, more specific means, which we do in two different ways.

- The formalistic answer is that in our proof of completeness we only use the Hoare rules mentioned in §5.4 and 5.6, structural induction on terms, and induction on the number of methods and the number of uninitialized classes (both of which are finite). All remaining steps are term rewriting and finitary predicate-logical derivations (involving *e.g.* the rules of the operational semantics in order to derive the postcondition of the MGF), and in particular we do not employ rule induction.

- We additionally give a more constructive answer: a recursive procedure, inspired by our proof of completeness. It generates a proof outline for any (valid) goal $\Gamma,\emptyset\vdash t$, assuming that Γ is well-structured. We sketch the procedure (glossing over applications of the rules *conseq* and *Xcpt*) and argue informally that this procedure terminates and thus one can conclude that the proof is finitary.

  Assume that $M = \{(C{,}sig).\ \mathsf{is\_methd}\ \Gamma\ C\ sig\}$ is the set of methods in Γ, the function $spec = \lambda tf\ (C{,}sig).\ \{(Pre\ C\ sig)\}\ tf\ C\ sig{-}\!\succ\ \{Post\ C\ sig\}$ forms a triple with the (most general) method specification, $MSpecs = spec\ \mathsf{Methd}\ ``\ M$ abbreviates the set of all method specifications, and $specb = spec\ (\mathsf{body}\ \Gamma)$ gives the body of a method with its specification. Note that $M$ and thus also $MSpecs$ and $specb\ ``\ M$ are finite.

  For showing $\Gamma,\emptyset\vdash t$ we use the following initial proof tree.

$$\dfrac{\dfrac{\dfrac{}{\Gamma{,}MSpecs|\!\vdash MSpecs}\,asm}{\Gamma{,}MSpecs\vdash specb\ m_1}\ {\scriptstyle Methd\_asm(MSpecs,\, specb\ m_i)}\ \dots\ \dfrac{\dfrac{}{\Gamma{,}MSpecs|\!\vdash MSpecs}\,asm}{\Gamma{,}MSpecs\vdash specb\ m_{|M|}}}{\dfrac{\dfrac{\Gamma{,}MSpecs\vdash specb\ ``\ M}{\Gamma{,}\emptyset|\!\vdash MSpecs}\,Methd}{\Gamma{,}\emptyset\vdash t}\,Methd\_asm(\emptyset,\, t)}\ |M|{*}insert{+}empty$$

$Methd\_asm(A,t)$ is a proof procedure used for showing $\Gamma{,}A|\!\vdash MSpecs \longrightarrow \Gamma{,}A\vdash t$. The procedure assumes $\Gamma{,}A|\!\vdash MSpecs$ and calls the recursive procedure $prove\_triple(t)$. (One could alternatively use the rule *cut* and a proof procedure $Methd\_asm'(A,t)$ proving $\Gamma{,}A{\cup}MSpecs\vdash t$.) We define $prove\_triple(\{P\}\ t\succ\ \{Q\})$ by case distinction on t.

case $\mathsf{Methd}\ C\ sig$**:** use *weaken* and the assumption.

case $\mathsf{init}\ C$**:** prove $\Gamma{,}A\vdash\{P\ \wedge.\ \mathsf{initd}\ C\}\ t\succ\ \{Q\}$ using contradiction or *Done*; prove $\Gamma{,}A\vdash\{P\ \wedge.\ \mathsf{Not}\circ\mathsf{initd}\ C\}\ t\succ\ \{Q\}$ trying contradiction first and resort to *Init* plus recursion only if unsuccessful.

case $\forall b.\ \Gamma{,}A\vdash\{P'{\leftarrow}{=}b\}\ tf\ (\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2){\succ}\ \{Q'\}$**:** expand to both cases True and False and proceed recursively.

case $\forall D\ l.\ \Gamma{,}A\vdash\{R'\ D\ l\}\ \mathsf{Methd}\ D\ sig{-}\!\succ\ \{S'\ l\}$**:** take a fixed but arbitrary D, exploit the precondition $R'\ D\ l$ in order to restrict the variety of D, and use *weaken* and the assumption.

all other universal quantifications: proceed recursively for any fixed but arbitrary value since the proof is uniform.

otherwise: use the canonical rule plus recursion.

The procedure $prove\_triple(t)$ terminates (assuming that the proofs of the emerging side conditions terminate) because for all recursive calls the number of quantifiers is reduced or the Java^{light} terms involved become structurally smaller, except for $\mathsf{init}\ C$ where the number of uninitialized classes is reduced.

5.8.6 Summary

We have proven

Theorem 3 *For well-structured programs our axiomatic semantics is relatively complete.*

The proof is non-trivial because next to structural induction it requires nested auxiliary inductions for handling mutual recursion and class initialization. The MGF approach has been successfully refined and applied. Here well-formedness and well-typedness only play a minor role.

Chapter 6

Example

In the previous chapters we have introduced both our model of the Java language and proofs on the language in general. Although our focus is clearly on meta-theory, in this chapter we give a small application example. It should

- give an impression of the features covered by the model
- illustrate our approach and the interplay of the concepts introduced
- provide a rudimentary way of validating the model
- demonstrate the suitability for actual program verification

This list of goals also guides the structure of the present chapter.

6.1 Program

Subsequently we handle the following example program fragment.

```
interface HasFoo {
  public Base foo(Base z);
}

class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

class Ext extends Base {
  int vee;
  public Ext foo(Base z) {
    ((Ext) z).vee = 1;
    return null;
  }
}
```

```
Base e = new Ext();
try {
  e.foo(null);
}
catch(NullPointerException z) {
  while(Ext.arr[2]) ;
}
```

The program fragment consists of three simple but complete type declarations and a block of statements that might occur in any method body that has access to these declarations, for instance in the main method of a test class:

```
class Example {
  public static void main(String args[]) throws Throwable {

  }
}
```

Note that the result type of method `foo` within class `Ext` is more specific than for the overridden method in `Base`. We use this to demonstrate that useful generalizations (*cf.* §1.5.3) are actually possible even if current Java does not allow them. In order to compile this example program with any current Java implementation, one has to replace the result type `Ext` by `Base`.

The program does not intend to give an example of good programming style or common patterns in software development, but should give a condensed illustration of as many features of Java$^{\ell ight}$ as possible. It includes:

- class, array and interface declarations with inheritance, hiding of fields, and overriding of methods (with refined result type),
- primitive types, reference types and their relations,
- method calls with dynamic binding, parameter and return value passing and access,
- local variable declarations, assignment, and access,
- expression statements, sequential composition, loops, literal values, field assignment and type casts,
- exception generation and propagation, `try _ catch` and `throw` statements.
- static initialization and dynamic instance creation

6.2 Model

We translate the program fragment — currently without tool support — to an Isabelle theory built on top of our formalization of Java$^{\ell ight}$. See the appendix for the actual code. In this chapter we adopt the special typographic convention that logical entities declared only for the sake of the example are printed in typewriter font and are not listed in the index.

6.2.1 Names

We have to represent the Java type names `HasFoo`, `Base`, and `Ext` as members of the TName alternative of the HOL type *tname* and state that they are pairwise different. In order to avoid the clutter when explicitly using the injection TName, we employ syntax translations mapping the names to corresponding internal names prepended by TName. Introducing the internal names as members of the type *tnam*, we encounter a minor technical problem: Due to limitations of Isabelle/HOL, the unspecified type *tnam* (which actually should be a parameter of the theory `Name`) cannot simply be instantiated as required. Our workaround is to define an extra free datatype *tnam_* containing the internal names as its constructors and asserting *tnam_* to be isomorphic to *tnam*.

The field and variable names `arr`, `vee`, `z`, and `e` that should be of type *ename* are dealt with analogously.

6.2.2 Method Declarations

We first model the three different declarations of method `foo`.

- Within interface `HasFoo`, the declaration consists of the signature and the method head. The former contains the name `foo` of type *mname* and the single parameter type Class `Base`, and the latter contains the flag indicating a non-static method, the parameter name `z`, and the result type Class `Base`.
- Within class `Base`, the declaration consists of the information just given plus a method body, which here is just the empty list of local variables, the empty body Skip, and the return expression !!`z` accessing the parameter `z`.
- Within class `Ext`, the signature and the method head is the same except that the result type is Class `Ext`. Here the list of local variables is empty again, the body consists of a single expression statement, and the result expression is the Null literal. The expression statement is an assignment of the integer literal `1` to the `vee` field variable of the object referenced by the parameter `z` which is cast to type Class `Ext`. Due to this type cast, the field name `vee` refers to the field (of type PrimT `int`) defined in class `Ext`, such that the field variable is annotated with `Ext` as the defining class and False indicating a non-static field.

```
HasFoo_foo ≡ ((foo,[Class Base]),  (False,[z],Class Base))
Base_foo   ≡ ((foo,[Class Base]), ((False,[z],Class Base),([],Skip,!!z)))
Ext_foo    ≡ ((foo,[Class Base]), ((False,[z],Class Ext ),
                 ([],Expr({Ext,False}Cast (Class Ext) !!z..vee := Lit (Intg 1)),Lit Null)))
```

6.2.3 Class and Interface Declarations

Using the above definitions, we can model the declarations of the three reference types in the program.

- The model of interface `HasFoo` has an empty list of superinterfaces and a singleton list of method declarations containing the entry for the first declaration of method `foo` given above.
- Our model of class `Base` explicitly gives `Object` as the superclass and `HasFoo` as the only implemented interface. The list of fields contains two entries, one for `arr`, where the inner pair of values means that it is a static array of Boolean values, and one for `vee` which is a non-static field of interface type `HasFoo`. There is one method, *viz.* the

second variant of method `foo`. The static initializer is an assignment (as an expression statement) of an array creation expression to the field variable `arr_viewed_from Base`. The annotations of this field variable indicate that it is static and has been declared in class `Base`, and the reference expression of the field is the static class name `Base`, or to be exact, its emulation using StatRef as defined in §2.4.2. In the array creation expression the element type is `boolean`, and the size is given by the literal integer `2`.

- The model of class `Ext` mentions `Base` as the superclass, the empty list of implemented interfaces, a new entry for field `vee`, this time of type PrimT `int`, the third version of method `foo` (overriding the one of the superclass) and the default static initializer Skip.

```
arr_viewed_from C ≡ {Base,True}StatRef (ClassT C)..arr
HasFooInt ≡ ([],[HasFoo_foo])
BaseCl    ≡ (Object, [HasFoo],
              [(arr, (True, PrimT boolean[])),
               (vee, (False,Iface HasFoo    ))],
              [Base_foo],
              Expr(arr_viewed_from Base := new (PrimT boolean)[Lit (Intg 2)]))
ExtCl     ≡ (Base  , [],
             [(vee, (False,PrimT int))],
             [Ext_foo],
             Skip)
```

6.2.4 Test Program

- Our test program `tprg` consists of the list with the single interface (`HasFoo`,`HasFooInt`) and the list of the two user-defined classes (`Base`,`BaseCl`) and (`Ext`,`ExtCl`) together with the language-defined standard classes.
- The code fragment `test` with parameter pTs is the sequential composition of the (initializing) assignment of a new instance of class `Ext` to the local variablee and a `try _ catch _` statement. The `try` clause is a method call (as an expression statement) relative to `e` of the method `foo` with the singleton parameter list consisting of the Null literal. The call is annotated with the indicator IntVir for dynamic method lookup and two times with class `Base`, namely as the static type of the reference expression !!`e` and as the defining class of the method `foo`. We have abstracted from the remaining type annotation for the parameter type list of `foo`, which will turn out to be the singleton list consisting of class `Base`. The `catch` clause has the exception parameter `z` of type `NullPointer`, and its body is a `while` loop with the empty body Skip. The loop condition is the access to an element of the array referred to by the field variable `arr_viewed_from Ext` at the index given by the integer literal `2`.

```
ifaces    ≡ [(HasFoo,HasFooInt)]
classes   ≡ [(Base,BaseCl),(Ext,ExtCl)] @ standard_classes
tprg      ≡ (ifaces,classes)
CTBase    ≡ ClassT Base
test pTs  ≡ e :== new Ext;
            try Expr({CTBase,CTBase,IntVir}!!e..foo({pTs}[Lit Null]))
            catch((SXcpt NullPointer) z)
              (while(Acc (Acc (arr_viewed_from Ext)[Lit (Intg 2)])) Skip)
```

The sequence of statements `test` could have been embedded in `tprg`, which we have left out for simplicity.

6.3 Properties

We now aim to prove the characteristic properties of our example program expected also by manual code inspection, by the compiler and by a test run of the program. The ability of deriving those properties gives confidence that our model of the Java language in general and the test program in particular is adequate.

6.3.1 Well-formedness

We begin the analysis by showing that `tprg` is well-formed, a static property checked by the compiler as well.

First we have to show that `tprg` is well-structured since this property is required for performing method and field lookup. We show that the declarations of class `Object`, class `Throwable`, the remaining standard exception classes, class `Base`, class `Ext`, and interface `HasFoo` are all well-structured, where the main effort is to show that various subclass and subinterface relations do not hold. Now it is easy to conclude that ws_prog `tprg` holds.

Exploiting well-structuredness we can derive the equations

```
fields tprg Base = [((arr, Base), (True,  PrimT boolean[])),
                    ((vee, Base), (False, Iface HasFoo     ))]
fields tprg Ext  = [((vee, Ext ), (False, PrimT int))] @ fields tprg Base

cmethd tprg Object = empty
cmethd tprg Base   = empty((foo,[Class Base])↦(Base, (False,[z],Class Base), [], Skip, !!z))
cmethd tprg Ext    = cmethd tprg Base ++
                       empty((foo,[Class Base])↦(Ext , (False,[z],Class Ext ), [],
                         Expr ({Ext,False}Cast (Class Ext) !!z..vee:=Lit (Intg 1)),
                         Lit Null))
```

where we assume that class `Object` does not have methods:

```
Object_mdecls ≡ []
```

Independently of well-structuredness we can derive that the set of classes and interfaces is unique, that the subinterface relation is empty, and that the following subclass relations hold:

```
tprg⊢SXcpt xn ≼c SXcpt Throwable
tprg⊢Ext ≼c Base
```

Finally, by unfolding the respective definitions, exploiting well-structuredness and applying the rules for well-typedness to the terms in the method bodies, we can show well-formedness of all declared elements, in particular

```
wf_mdecl tprg Base Base_foo
wf_mdecl tprg Ext  Ext_foo

wf_cdecl tprg (Base,BaseCl)
wf_cdecl tprg (Ext ,ExtCl  )

wf_prog tprg
```

where we further assume that the standard exceptions do not have methods:

```
SXcpt_mdecls ≡ []
```

6.3.2 Well-typedness

In order to demonstrate the effect of the typing rules we show that `test` $?pTs$ is well-typed. The argument $?pTs$ is a schematic variable (standing for the type annotation of the method call that we left open) which we expect to get suitably instantiated during the proof.

Our proof obligation is

(tprg, empty(EName e$\mapsto$Class Base))$\vdash$test $?pTs$::$\surd$

where the local type environment consists of one entry for the local variable e of class `Base`.

The proof proceeds by applying about 20 instances of the typing rules (in backward-chaining manner) and about 15 applications of term rewriting for expansion of definitions and simplification.

Local Variables

After a few steps, we have to prove the subgoal

(tprg, empty(EName e$\mapsto$Class Base))$\vdash$LVar (EName e)::=$?T4$

and after applying the rule for local variables (*cf.* §2.8.6) we have to show

snd (tprg, empty(EName e$\mapsto$Class Base)) (EName e) = Some $?T4$

which holds (instantiating $?T4$ to Class `Base`) because of the entry for e in the type environment.

Method Call

A few steps later we arrive at

(tprg, empty(EName e$\mapsto$Class Base))$\vdash${CTBase,CTBase,IntVir}!!e..foo({$?pTs$}[Lit Null])::−$?T8$

and after applying the rule for method calls (*cf.* §2.8.5) we have to show

max_spec tprg (CTBase) (foo, [NT]) = {((CTBase, $?m8$, $?pns8$, $?T8$), $?pTs$)}

From the lemma

appl_methds tprg (CTBase) (foo, [NT]) = {((CTBase, (False,[z],Class Base)), [Class Base])}

we obtain the following subgoal, which we solve by reflexivity:

$?m8$ = False $\wedge$ $?pns8$ = [z] $\wedge$ $?T8$ = Class Base $\wedge$ $?pTs$ = [Class Base]

Note that at this point the schematic variable $?pTs$ is instantiated and thus the missing type annotation has been computed, as done also by the compiler.

Field Variables

Again a few steps later we arrive at

(tprg, empty(EName e$\mapsto$Class Base)(EName z$\mapsto$Class (SXcpt NullPointer)))$\vdash$
arr_viewed_from Ext::=PrimT boolean[]

where due to the rule for the `try _ catch _` statement (*cf.* §2.8.3) the local type environment has been augmented by an entry for the exception parameter `z` of class `NullPointer`. After applying the rule for field variables (*cf.* §2.8.6) we have to show

cfield tprg Ext arr = Some (Base, True, PrimT boolean[])

which we do by rewriting with the definition of cfield and exploiting the above equations for fields tprg Ext and fields tprg Base. Note that even if `arr` is referred to via class `Ext`, we get (through inheritance) the static field declared in class `Base`.

The remainder of the proof is straightforward.

6.3.3 Symbolic Execution

A good validation test for our formalization and final demonstration of its use is symbolically executing the test program using the operational semantics. Doing so we noticed a slip in an earlier version of our model of class objects: their representations, as governed by the function var_tys introduced in §3.1.1, contained superfluous entries for inherited static fields even though such fields are shared among all subclasses of the defining class.

We execute the program fragment test [Class Base] within program tprg from the start state Norm s_0 where s_0 = st empty empty under the assumption that there are at least four free locations on the heap of s_0. To capture the not yet known final state we use the schematic variable $?\sigma_9$. Thus our goal reads as

$$\mathsf{atleast\_free}\ (\mathsf{heap}\ s_0)\ 4 \longrightarrow \mathtt{tprg}\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{test}\ [\mathsf{Class}\ \mathtt{Base}]} ?\sigma_9$$

In other words, we simulate program execution by a proof of the above formula within Isabelle/HOL, leaving the final state initially undetermined. Henceforth we mention assumptions only where necessary for understanding.

The proof of this goal involves about 45 applications of rules of the operational semantics, 35 applications of the simplifier rewriting with numerous equations (typically on state transformers), and a few applications of predicate-logical reasoning to deal with side conditions of the rules. The proof is rather syntax-directed and straightforward. On the other hand, it is quite detailed — just note the number of on-the-fly abbreviations we introduce below — and requires non-trivial properties of the heap, used as assumptions and derived facts.

Class Initialization and Instance Creation

After a few steps we get to the subgoal

$$\mathtt{tprg}\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathtt{new\ Ext} -\succ ?v9} ?\sigma_{23}$$

which, applying the rule for class instance creation (*cf.* §3.2.8), leads us further to

$$\mathtt{tprg}\vdash \mathsf{Norm}\ s_0 \xrightarrow{\mathsf{init}\ \mathtt{Ext}} ?\sigma_2$$

Since in the current state Norm s_0 there is no entry for the class object of class Ext, it has to be initialized. This in turn triggers initialization of class Base, which in turn triggers initialization of class Object, which is trivial. The static initializer of class Base is non-trivial, and thus we have to execute

$$\mathtt{tprg}\vdash \mathsf{Norm}\ s_1 \xrightarrow{\mathsf{Expr}\ (\mathtt{arr\_viewed\_from\ Base\ :=\ new}\ \mathsf{PrimT}\ \mathtt{boolean}[\mathsf{Lit}\ (\mathsf{Intg}\ 2)])} ?\sigma_2'$$

where

$$\begin{aligned} \mathit{globs1} = \mathsf{empty}&(\mathsf{Stat}\ \mathtt{Object} \mapsto (\mathsf{arbitrary},\ \mathsf{empty}))\\ &(\mathsf{Stat}\ \mathtt{Base} \quad \mapsto (\mathsf{arbitrary},\ \mathsf{empty}(\mathsf{Inl}\ (\mathtt{arr},\ \mathtt{Base}) \mapsto \mathsf{Null})))\\ &(\mathsf{Stat}\ \mathtt{Ext} \quad\ \mapsto (\mathsf{arbitrary},\ \mathsf{empty}))\\ s_1 = \mathit{globs1}&\ \mathsf{empty} \end{aligned}$$

globs1 contains the class objects of the classes Object, Base and Ext whose fields have been initialized to their default values. The static field variable arr of class Base, which has been initialized to the Null reference, is now updated with an array object that has to be allocated on the heap. Consequently, after a few further steps applying the rule for array creation (*cf.* §3.2.8) we arrive at

$$\mathtt{tprg}\vdash \mathsf{Norm}\ s_1 \xrightarrow{\mathsf{halloc\ Arr}\ (\mathsf{PrimT}\ \mathtt{boolean})\ 2 \succ ?a'} ?\sigma_{12}$$

The halloc relation can be successfully fulfilled only if

$$\exists a.\ \mathsf{new\_Addr}\ (\mathsf{heap}\ s_1) = \mathsf{Some}\ a \land \mathsf{atleast\_free}\ (\mathsf{heap}\ s_1)\ 2$$

holds. This can indeed be derived from our assumption $\mathsf{atleast\_free}\ (\mathsf{heap}\ s_0)\ 4$. Assuming that $\mathsf{new\_Addr}\ (\mathsf{heap}\ s_1)$ yields location a we obtain

$$\begin{aligned} ?\sigma_{12}{}' &= \mathsf{Norm}\ (\mathsf{st}\ \mathit{globs12}\ \mathsf{empty}) \\ ?\sigma_2' &= \mathsf{Norm}\ (\mathsf{st}\ \mathit{globs2}\ \ \mathsf{empty}) \\ ?\sigma_2 &= ?\sigma_2' \end{aligned}$$

where

$$\begin{aligned} \mathit{globs12} &= \mathit{globs1}\ (\mathsf{Heap}\ a\ \ \mapsto \mathit{obj\_a}) \\ \mathit{globs2} &= \mathit{globs12}(\mathsf{Stat}\ \mathtt{Base}{\mapsto}(\mathsf{arbitrary},\ \mathsf{empty}(\mathsf{Inl}\ (\mathtt{arr},\ \mathtt{Base}){\mapsto}\mathsf{Addr}\ a))) \\ \mathit{obj\_a} &= (\mathsf{Arr}\ (\mathsf{PrimT}\ \mathtt{boolean})\ 2,\ \mathsf{empty}(\mathsf{Inr}\ 0{\mapsto}\mathsf{Bool}\ \mathsf{False})(\mathsf{Inr}\ 1{\mapsto}\mathsf{Bool}\ \mathsf{False})) \end{aligned}$$

This finishes the initialization of class `Ext`. Assuming that the new instance of class `Ext` is allocated at location b, the intermediate state after assigning the object reference to e and before executing the `try _ catch _` statement is

$$\sigma_3 = \mathsf{Norm}\ s_3$$

where

$$\begin{aligned} s_3 &= \mathsf{st}\ \mathit{globs3}\ \mathit{locs3} \\ \mathit{locs3} &= \mathsf{empty}(\mathsf{EName}\ \mathtt{e}{\mapsto}\mathsf{Addr}\ b) \\ \mathit{globs3} &= \mathit{globs2}(\mathsf{Heap}\ b{\mapsto}\mathit{obj\_b}) \\ \mathit{obj\_b} &= (\mathsf{CInst}\ \mathtt{Ext},(\mathsf{empty}(\mathsf{Inl}\ (\mathtt{vee},\ \mathtt{Base}){\mapsto}\mathsf{Null}\ \ ) \\ &\qquad\qquad\qquad (\mathsf{Inl}\ (\mathtt{vee},\ \mathtt{Ext}\ ){\mapsto}\mathsf{Intg}\ 0))) \end{aligned}$$

Note that within $\mathit{obj\_b}$ there are the two different versions of field `vee` as declared in the classes `Base` and `Ext`, initialized to their respective default values.

At this stage we can derive from our assumptions that

$$\mathsf{atleast\_free}\ (\mathsf{heap}\ s_0(a{\mapsto}\mathit{obj\_a})(b{\mapsto}\mathit{obj\_b}))\ 2 \land a \neq b$$

holds, which we will need later. Note that even if we reason in an operational style here, we have to collect propositions, as typically done for program verification using Hoare logic. Moreover, the rather low-level reasoning forces us to deal with uninteresting details such as distinctness of certain memory locations.

Method Call

The next important step is to evaluate the method call

$$\mathtt{tprg}{\vdash}\sigma_3 \xrightarrow{\{\mathsf{CTBase},\mathsf{CTBase},\mathsf{IntVir}\}!!\mathtt{e}..\mathtt{foo}(\{[\mathsf{Class\ Base}]\}[\mathsf{Lit\ Null}]){-}{\succ}?v7} ?\sigma_7$$

After evaluating the target reference to $\mathsf{Addr}\ b$ and the argument list to [Null], the dynamic type of the reference $\mathsf{target}\ \mathsf{IntVir}\ s_3\ (\mathsf{Addr}\ b)\ (\mathtt{CTBase})$ evaluates to the tag of $\mathit{obj\_b}$, which is `Ext`. Thus method `foo` of class `Ext` is invoked

$$\mathtt{tprg}{\vdash}\sigma_4 \xrightarrow{\mathsf{Methd}\ \mathtt{Ext}\ (\mathtt{foo},\ [\mathsf{Class\ Base}]){-}{\succ}?v8} ?\sigma_6{}'$$

from state

$$\sigma_4 = \mathsf{Norm}\ s_4$$

where

$$\begin{aligned} s_4 &= \mathsf{st}\ \mathit{globs3}\ \mathit{locs8} \\ \mathit{locs8} &= \mathsf{empty}(\mathsf{EName}\ \mathtt{z}{\mapsto}\mathsf{Null})(\mathsf{This}{\mapsto}\mathsf{Addr}\ b) \end{aligned}$$

Here the local variables consist of the parameter `z` bound to Null and This bound to the calling reference.

Field Variables

After unfolding the method body and a few further steps (including two calls to init Ext that in the current state σ_4 are equivalent to Skip) we get to

$$\texttt{tprg}\vdash\sigma_4 \xrightarrow{\{\texttt{Ext},\mathsf{False}\}\mathsf{Cast}\ (\mathsf{Class}\ \texttt{Ext})\ !!\texttt{z..vee}{=}{\succ}(?w6,\ ?f6)} ?\sigma_6$$

Evaluating fvar Ext False vee (the (locals s_4 (EName z))) $s4'$ (due to the rule for field variables, *cf.* §3.2.10), we obtain the exceptional state $?\sigma_6 =$ (Some (StdXcpt NullPointer), s_4) because z contains the Null reference. Thus the state update function *?f6* and the value *?v8* (as well as the field value *?v6*, which is used for reading access only anyway) turn out to be irrelevant. Due to the exception the evaluation of the right-hand side Lit (Intg 1) of the assignment and the result expression Lit Null is short-circuited, and the method call returns with the NullPointer exception thrown (and of course the former local variables of the callee restored): $?\sigma_7$ =(Some (StdXcpt NullPointer), s_3).

Try & Catch Statement

Now the catch clause is going to be executed. Before we can do this a proper exception object of class NullPointer has to be allocated, as done by the relation sxalloc (*cf.* §3.2.5):

$$\texttt{tprg}\vdash ?\sigma_7 \xrightarrow{\mathsf{sxalloc}} ?\sigma_{78}$$

We make use of the derived fact atleast_free (heap $s_0(a{\mapsto}obj\_a)(b{\mapsto}obj\_b)$) 2, and assuming that the next fresh location returned by new_Addr is c we obtain

$?\sigma_{78} =$ (Some (XcptLoc c), st *globs8 locs3*)
where
globs8 = *globs3*(Heap $c{\mapsto}obj\_c$)
obj_c = (CInst (SXcpt NullPointer),empty)

and can further derive $a \neq c$. The body of the catch clause is now executed from the state

$\sigma_8 =$ Norm s_8
where
s_8 = st *globs8 locs8*
locs8 = *locs3*(EName z$\mapsto$Addr c)

that is, the exception parameter is bound to the location of *obj_c*.

Loop Statement

Finally, after a few steps of evaluating the while loop and the array access contained in it, evaluating *var* tprg (Intg 2) (Addr a) σ_8 — exploiting the facts $a \neq b$ and $a \neq c$ — yields the state (Some (StdXcpt IndOutBound), s_8) because the maximal index of the array *obj_a* is less than 2. Thus further execution of the loop is short-circuited and the final state is

```
?σ9 = (Some (StdXcpt IndOutBound),
        st (empty(Stat Object↦(arbitrary, empty))
                 (Stat Base   ↦(arbitrary, empty(Inl (arr, Base)↦Addr a)))
                 (Stat Ext    ↦(arbitrary, empty))
                 (Heap a↦obj_a)(Heap b↦obj_b)(Heap c↦obj_c))
           (empty(EName e↦Addr b)(EName z↦Addr c)))
```

Note that in this state the exception parameter z of the try _ catch _ statement is still present (but has become inaccessible).

6.3.4 Proof using Hoare Logic

In this section we use the rules of the axiomatic semantics to derive a non-trivial property of the example. We comment on the most interesting steps of the proof. This should give an impression of how the axiomatic semantics can be used for actual program verification. For demonstrating the power of a Hoare logic in general, other programs, namely those involving non-trivial loops and recursive method calls, would be surely suited better. Yet the current example has the advantage of involving almost all features of Java^{light} and allowing direct comparison with the just given symbolic program execution using the operational semantics.

We show that the test program terminates (if at all — this is partial correctness) abruptly with an `IndOutBound` exception:

$\texttt{tprg},\emptyset\vdash\{Pre\}$ `.test` [Class `Base`]. $\{\lambda Y\,\sigma\,Z.\ \mathsf{fst}\ \sigma = \mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{IndOutBound})\}$

where

$Pre \equiv \mathsf{Normal}\ (\lambda Y\,\sigma\,Z.\ \mathsf{heap\_free}\ 4\ \sigma \wedge \neg\mathsf{initd}\ \texttt{Base}\ \sigma \wedge \neg\mathsf{initd}\ \texttt{Ext}\ \sigma)$

$\mathsf{heap\_free}\ n \equiv \lambda\sigma.\ \mathsf{atleast\_free}\ (\mathsf{heap}\ (\mathsf{snd}\ \sigma))\ n$

That is, we aim to derive from the empty set of assumptions that if initially there are at least four free locations on the heap and the classes `Base` and `Ext` are not initialized then after termination of the program the exception `IndOutBound` has been thrown. This property relies on a bunch of more or less implicit properties of the program control involving class initialization, dynamic binding, and actual values of method parameters. One could of course prove also other properties like `((Ext)e).vee == 0`.

We make our way through the control flow as directed by the syntax in the usual "backwards" style where the current postcondition is fully known and directs the instantiations of schematic variables typically contained in the precondition mostly automatically. That is, we can apply the syntax-directed Hoare logic rules mostly without making use of the rule of consequence and explicitly instantiating assertions. We have reached this desirable convenience by designing the rules such that the postcondition of the triple in their consequent consists (typically) solely of a free assertion variable that can always be instantiated as required in the application. Yet in a few places explicit instantiations are needed, and sometimes we deliberately use them in order to manually simplify assertions.

This example proof takes about 130 steps, 50 of which are applications of syntax-directed Hoare logic rules. We apply the rule of consequence 13 times and do about 20 explicit instantiations of schematic assertion variables. The simplifier or classical reasoner (or their combination) is called about 40 times. One third of the proof deals with class initialization.

Try Rule

We first have to show that the `catch` clause at the end of the program is actually taken. We instantiate the assertion Q of the *Try* rule (*cf.* §5.6.3) with

$\lambda Y\,\sigma\,Z.\ \texttt{arr\_inv}\ (\mathsf{snd}\ \sigma) \wedge \texttt{tprg},\sigma\vdash\mathsf{catch}\ \mathsf{SXcpt}\ \texttt{NullPointer}$

where $\texttt{arr\_inv} \equiv \lambda s.\ \exists obj\ a\ T\ el.\ \mathsf{heap}\ s\ a = \mathsf{Some}\ (\mathsf{Arr}\ T\ 2, el) \wedge$
$\qquad\qquad \mathsf{globs}\ s\ (\mathsf{Stat}\ \texttt{Base}) = \mathsf{Some}\ obj \wedge \mathsf{snd}\ obj\ (\mathsf{Inl}\ (\texttt{arr},\ \texttt{Base})) = \mathsf{Some}\ (\mathsf{Addr}\ a)$

stating that immediately before the `catch` clause a `NullPointer` exception has been thrown and the property `arr_inv` holds. This property will be needed in the condition of the `while` loop and will take part in the assertions as an invariant all the way back to the creation of the static array `arr`. It states that there is a class object for class `Base` with a field named `arr` which points to an array on the heap with no more than two components. Note that we do not have to say anything about the type of the array components or about other global or local entities possibly contained in the state, such abstraction being one of the main advantages of a Hoare logic in general.

Loop Rule

After a few steps of preparation we are ready to apply the rule *Loop* (*cf.* §5.6.2):

$$\begin{array}{l} \texttt{tprg},\emptyset\vdash\{\mathsf{Normal}\ (\lambda Y\ \sigma\ Z.\ \texttt{arr\_inv}\ (\mathsf{snd}\ s))\} \\ \qquad .\texttt{while}\ (\mathsf{Acc}\ (\mathsf{Acc}\ (\texttt{arr\_viewed\_from}\ \texttt{Ext}[\mathsf{Lit}\ (\mathsf{Intg}\ 2)]))\ \mathsf{Skip}. \\ \qquad \{(\lambda Y\ \sigma\ Z.\ \mathsf{fst}\ \sigma = \mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{IndOutBound}))\leftarrow{=}\mathsf{False}{\downarrow}{=}\bullet\} \end{array}$$

The loop body can be dealt with by contradiction between the normal pre-state expected for it and the exceptional state actually present. After a few further steps we reach the reason of the `IndOutBound` exception being thrown: accessing the array `Ext.arr` at index `2`.

$$\begin{array}{l} \texttt{tprg},\emptyset\vdash\{\mathsf{Normal}\ (\lambda Y\ \sigma\ Z.\ \texttt{arr\_inv}\ (\mathsf{snd}\ \sigma))\}\ \texttt{arr\_viewed\_from}\ \texttt{Ext}{=}\succ\ \{\mathsf{Normal}\ (\lambda\mathsf{Var}\ (v,\ f)\ \sigma\ Z. \\ \qquad \mathsf{fst}\ (\mathsf{snd}\ (\mathsf{avar}\ \texttt{tprg}\ (\mathsf{Intg}\ 2)\ v\ \sigma)) = \mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{IndOutBound}))\} \end{array}$$

Method Call

Going further backwards, we have to show that `arr_inv` is maintained by allocating the exception object for the `NullPointer` exception and then augment the invariant by the proposition heap_free `2` indicating that there has been enough memory left to do that allocation. The current list of proof obligations meanwhile is

$$\begin{array}{l} \texttt{tprg},\emptyset\vdash\{\mathsf{Normal}\ ?P1\}\ \{\texttt{CTBase},\texttt{CTBase},\mathsf{IntVir}\}!!\texttt{e..foo}(\ \{[\mathsf{Class}\ \texttt{Base}]\}[\mathsf{Lit\ Null}]){-}\succ \\ \qquad \{((\lambda Y\ (x,s)\ Z.\ x = \mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{NullPointer})\ \wedge\ \texttt{arr\_inv}\ s)\ \wedge.\ \mathsf{heap\_free}\ 2)\leftarrow\bullet\} \\ \texttt{tprg},\emptyset\vdash\{\mathsf{Normal}\ (\lambda Y\ \sigma\ Z.\ \mathsf{heap\_free}\ 4\ \sigma\ \wedge\ \neg\ \mathsf{initd}\ \texttt{Base}\ \sigma\ \wedge\ \neg\ \mathsf{initd}\ \texttt{Ext}\ \sigma)\}\ .e\ \texttt{:==}\ \texttt{new}\ \texttt{Ext}. \\ \qquad \{\mathsf{Normal}\ ?P1\} \end{array}$$

where the precondition $\{\mathsf{Normal}\ ?P1\}$ is not — and does not need to be — fully instantiated yet. Tackling the first of these subgoals we apply a variant of the method call rule (*cf.* §5.6.8) that has been slightly simplified making use of fact that we already (statically) know that the dynamic type of the calling reference is `Ext`. The rule leaves us with the following four new subgoals:

$$\begin{array}{l} \texttt{tprg}\vdash\mathsf{IntVir}{\rightarrow}\texttt{Ext}{\preceq}\texttt{CTBase} \\ \forall a\ vs\ l.\ \texttt{tprg},\emptyset\vdash\{?R16\ a{\leftarrow}\mathsf{In3}\ vs\ \wedge.\ (\lambda\sigma.\ l = \mathsf{locals}\ (\mathsf{snd}\ \sigma))\ ;. \\ \qquad\qquad\qquad\quad \mathsf{init\_lvars}\ \texttt{tprg}\ \texttt{Ext}\ (\texttt{foo},\ [\mathsf{Class}\ \texttt{Base}])\ \mathsf{IntVir}\ a\ vs\} \\ \qquad\qquad\qquad\quad \mathsf{Methd}\ \texttt{Ext}\ (\texttt{foo},\ [\mathsf{Class}\ \texttt{Base}]){-}\succ\ \{\mathsf{set\_lvars}\ l\ .; \\ \qquad\qquad\qquad\quad ((\lambda Y\ (x,s)\ Z.\ x{=}\mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{NullPointer})\ \wedge\ \texttt{arr\_inv}\ s)\ \wedge.\ \mathsf{heap\_free}\ 2)\leftarrow\bullet\} \\ \forall a.\ \texttt{tprg},\emptyset\vdash\{?Q16{\leftarrow}\mathsf{In1}\ a\}\ [\mathsf{Lit\ Null}]\dot{=}\succ\{?R16\ a\ \wedge.\ (\lambda\sigma.\ \mathsf{obj\_class}\ (\mathsf{lookup\_obj}\ (\mathsf{snd}\ \sigma)\ a){=}\texttt{Ext})\} \\ \texttt{tprg},\emptyset\vdash\{\mathsf{Normal}\ ?P1\}\ !!\texttt{e}{-}\succ\ \{?Q16\} \end{array}$$

The first of these is immediate (unfolding its definition and exploiting the fact that within `tprg`, `Ext` is a subclass of `Base`). The second one is our first application example where we have to derive a triple that is explicitly universally quantified. We do this simply by deriving the triple for arbitrary but fixed a, vs and l. We continue by applying the rules *Methd* and then *thin* to throw away the assumption on Methd `Ext` (`foo`, [Class `Base`]) which we will not need. Then we unfold the method body and apply *Body*.

We handle the result expression Lit Null using the *Xcpt* rule because we know that an exception is thrown in the actual body, given next:

$$\begin{array}{l} \texttt{tprg},\emptyset\vdash\{?Q34\ a\ vs\ l\}\ .\mathsf{Expr}\ (\{\texttt{Ext},\mathsf{False}\}\mathsf{Cast}\ (\mathsf{Class}\ \texttt{Ext})\ !!\texttt{z..vee:=}\mathsf{Lit}\ (\mathsf{Intg}\ 1)). \\ \qquad \{(\mathsf{set\_lvars}\ l\ .;\ (\lambda Y\ (x,s)\ Z.\ x = \mathsf{Some}\ (\mathsf{StdXcpt}\ \texttt{NullPointer})\ \wedge\ \texttt{arr\_inv}\ s){\leftarrow}\bullet\ \wedge. \\ \qquad\ \ \mathsf{heap\_free}\ 2){\leftarrow}\mathsf{In1}\ \mathsf{arbitrary}\} \end{array}$$

After several canonical steps including application of the *FVar* rule, which is responsible for the generation of the exception, we have to handle the initialization statement init `Ext` of the class defining the method. At this stage it is advisable to manually instantiate *?R16* for which — after inspecting the current postcondition — we find the following value appropriate:

$\lambda a'$. Normal $((\lambda$Vals vs (x,s) Z. arr_inv s $\wedge$ inited Ext (globs s) $\wedge$ $a' \neq$ Null $\wedge$ hd vs = Null) $\wedge$.
heap_free 2)

Now since we are done with the method body, we return to the argument list of the call,

$\forall a.$ tprg,$\emptyset \vdash \{$*?Q16*$\leftarrow$In1 $a\}$ [Lit Null]$\doteq\succ$ $\{$*R16' a*$\}$

where the postcondition is abbreviated by *R16' a*. It depends on the universally quantified variable a, whereas *?Q16* only indirectly depends on a through the result substitution. Thus before we can apply *Cons*, we have to make the dependency explicit. In such situations, rules like

$$subst\_Val\ \frac{\forall v.\ \Gamma,A\vdash\{P'\ v\leftarrow \mathsf{Val}\ v\}\ t\succ\ \{Q\ v\}}{\forall v.\ \Gamma,A\vdash\{(\lambda w:.\ P'\ (\mathsf{the\_In1}\ w))\leftarrow\mathsf{Val}\ v\}\ t\succ\ \{Q\ v\}}$$

which partially instantiate *?Q16* are useful. After a few further steps the method call is done.

Object Creation

Since it remains to show

tprg,$\emptyset\vdash\{Pre\}$ .e:==new Ext. {Normal $((\lambda Y\ (x,s)\ Z.\ (\exists a.$ the (locals s (EName e)) = Addr a $\wedge$
obj_class (lookup_obj s a) = Ext) $\wedge$ arr_inv s) $\wedge$. initd Ext $\wedge$. heap_free 2)}

the next important step is to handle the creation of an instance of class Ext. With the auxiliary rule straightforwardly derived from the properties of object allocation

$$Alloc\ \frac{\begin{array}{c}\Gamma,A\vdash\{P\}\ t\succ\ \{\mathsf{Normal}\ (\lambda Y\ (x,s)\ Z.\ (\forall a.\ \mathsf{new\_Addr}\ (\mathsf{heap}\ s) = \mathsf{Some}\ a\longrightarrow\\ Q\ (\mathsf{Val}\ (\mathsf{Addr}\ a))\ (\mathsf{Norm}(\mathsf{init\_obj}\ \Gamma\ (\mathsf{CInst}\ C)\ (\mathsf{Heap}\ a)\ s))\ Z))\ \wedge.\ \mathsf{heap\_free}\ 2\}\end{array}}{\Gamma,A\vdash\{P\}\ t\succ\ \{\mathsf{Alloc}\ \Gamma\ (\mathsf{CInst}\ C)\ Q\}}$$

this does not cause any problems, despite the number of auxiliary functions like new_Addr involved. After a handful steps we obtain

tprg,$\emptyset\vdash\{Pre\}$ LVar (EName e)$=\succ$ $\{?P\}$
tprg,$\emptyset\vdash$\{Normal *?P*\} .init Ext. {Normal $((\lambda Y\ \sigma\ Z.$ arr_inv (snd σ) $\wedge$
vf = lvar (EName e) (snd σ)) $\wedge$. heap_free 3 $\wedge$. initd Ext)}

The first of these subgoals is trivial, while the second accounts for the remaining three dozen steps of the proof.

Static Initialization

Now we have to assume that class Ext and its superclass Base are not yet initialized. Initialization of Ext itself (*e.g.* using a straightforward variant of *Init* exploiting Ext $\neq$ Object) is immediate. Initialization of Base (using the same variant of *Init*) is a bit more involved — though not difficult — because its non-default static initializer Expr (arr_viewed_from Base := new PrimT boolean[Lit (Intg 2)]) has to be treated. This statement causes an extra call to init Base (due to the field variable contained in it) but this time no further initialization is needed because initialization of Base is already in progress. Finally, the class Object is potentially initialized. For this common simple case we have derived the auxiliary rule

$$triv\_init\_Object\ \frac{\mathsf{wf\_prog}\ \Gamma\qquad P\rightarrow(\mathsf{supd}\ (\mathsf{init\_class\_obj}\ \Gamma\ \mathtt{Object})\ .;\ P)}{\Gamma,A\vdash\{\mathsf{Normal}\ P\leftarrow\bullet\}\ .\mathsf{init}\ \mathtt{Object}.\ \{P\ \wedge.\ \mathsf{initd}\ \mathtt{Object}\}}$$

applicable for all P that ignore the class object for Object.

6.4 Summary

We have given an example program showing many features of our language model at work. Performing proofs on actual programs and symbolically executing them is tedious as one has to deal with a lot of detail and typically huge formulas (unless one introduces suitable abbreviations by hand, as we did). Serious applications, in particular program verification, would require more tool support like a (semi-)automatic verification condition generator and solver. For symbolic execution the tool by Berghofer [Ber00] that generates executable ML code from Isabelle/HOL theories promises to be helpful.

When comparing the proof by symbolic program execution (applying the operational semantics) and the proof using Hoare logic, in this case the former is slightly more direct because the assertions used with the Hoare logic introduce a certain notational overhead. Yet if a program to be verified contains proper iteration and recursion, the approach using Hoare logic will take full advantage of its built-in support for proving and exploiting loop invariants and method specifications, which would clearly be superior to a (low-level) proof using the operational semantics. The other important advantage of Hoare logic, namely the ability to formulate and manipulate properties with maximal abstraction, has been demonstrated even by our rather small example.

Chapter 7

Conclusions

In this chapter we summarize and evaluate our results, share some of our insights and point out advisable future work.

7.1 Achievements

We have formalized significant parts of the programming language Java and conducted meta-theory on it within the theorem prover Isabelle/HOL: a proof of type soundness and the development of a sound and complete Hoare logic. In particular, we have achieved the following contributions to the field of machine-checked programming language semantics.

Formalization of Java We have given the first rather comprehensive machine-checked (in the sense of fully formal) model of Java. Meanwhile, also a handful other formalizations within mechanical theorem proving systems exist, typically dealing with smaller fractions of the language. Attali et al. [ACR] give an executable specification covering more or less the whole language — including multi-threading — which on the other hand it is not suited (nor intended) for meta-theoretical proofs. By now no verification has been done using it.

Type Safety of Java We have proved that a large subset of Java is type-safe. Our proof goes further than others given so far in the sense that it is machine-checked (and thus maximally reliable) and covers not only full exception handling, but also static methods and fields and their initialization.

Extensions of Hoare Logic We have developed techniques for modeling expressions with side-effects within a Hoare Logic first-class, *i.e.* without assignments to intermediate variables, and for generally describing dependence of program terms upon values computed. We have further strengthened the rule of consequence and improved the handling of mutual recursion. By a non-trivial example we have demonstrated the suitability of our logic for actual program verification.

Soundness and Completeness We have shown that when defining and proving correct a method call rule in the presence of dynamic binding, type safety plays a crucial role. Moreover, we have given the first proof of completeness at all for an axiomatic semantics of an object-oriented language.

Maturity of Theorem Proving Technology We explored — and contributed to — the maturity of a state-of-the art theorem prover, Isabelle/HOL, concerning its use for both specifying realistic programming languages and proving some important non-trivial properties. We have demonstrated that for an experienced user such work is indeed feasible. Doing so we encountered only minor restrictions and difficulties using the system, yet noticed that for large-scale applications some *proof engineering* (in analogy to software engineering) facilities like enhanced support for modularity and change management are desirable.

7.2 Experience

Conducting the work described in this thesis has given insights into several topics, including

Java The design of (the most important parts of) Java is sound in the sense that it enjoys type safety. At several places, in particular concerning the relations between method return types, the language could be less restrictive. Array covariance and the mechanism for resolving static method overloading are perhaps not really worth the difficulties they introduce.

The Java specification is quite well-structured and unambiguous. There are just a few omissions, concerning exceptions and class initialization, and misleading statements, concerning method calls. Furthermore, we spotted and reported a few dozens linguistic, typographic, and other minor mistakes.

Part of these observations are new, and part of them had already been found by others and have just been confirmed here. When further extending the subset of Java covered, we expect to detect more problematic issues, since it is already known that for example the concurrency model and class loading contains flaws.

Axiomatic Semantics The completeness result for our Hoare logic implies that programs using even enhanced features like mutual recursion and dynamic binding can be proved correct. Yet, unfortunately, many of the rules given are quite complex and thus apparently not easy to use. This is in part due to our semantical notion of assertions better suited for meta-theoretical investigations than for practical use, but the main point is that Java is an inherently difficult language, taking into account *e.g.* mutual recursion, dynamic binding, exception handling, and static initialization.

As experience with our small but non-trivial example reveals, verifying programs heavily dealing with exceptions and class initialization is tedious, though further machine support would probably be a relief. Both our rather “deep” (*cf.* §1.6.5) formalization and Isabelle itself as the underlying theorem proving system is tailored more towards meta-theory rather than proofs on concrete systems of realistic size. Thus large-scale program verification would require adaptations of the model and extensions to the user interface and proof management, or possibly even transfer of the Hoare logic rules to a specialized integrated program development and verification tool.

After all, using an axiomatic semantics like ours for program verification helps to concentrate on the interesting properties of a program (rather than fiddling with details of the state as with an operational semantics) and provides powerful tools for dealing with loops and recursion. This general experience carries over from procedural to object-oriented languages like Java.

Larger-scale Formal Systems Since the subset of Java we chose contains most of the features of a real world programming language, our work was a major undertaking, also because currently the theorem prover technology that we could apply and the methodology of its use has been developed only recently and is still evolving. Even though both the formalization and the proofs have been kept as dense as reasonably possible, the system consists of almost 2000 lines of Isabelle theories (excluding comments) and more than 5000 lines of proof scripts. The table below gives more detailed approximate numbers.

	Number of Lemmas	Lines of Theory	Lines of Proof
Static Semantics	180	710	1220
Dynamic Semantics	120	430	780
Type Safety[1]	100	50	890
Axiomatic Semantics	160	490	1790
Example and Other	100	220	750
Total	660	1900	5430

Processing all theories and proofs on a machine considered passably fast these days takes about 40 minutes.

Our system is one of the largest applications of Isabelle developed so far. Without applying the simplification techniques listed in the introduction, developing it would have been much harder if not impossible for a single person in reasonable time. Due to important design decisions such as using an evaluation semantics, we believe that our model has become the most elegant and concise one available. This is also the experience of Büchi who chose our system as the basis for proving type safety of his extension of Java with Compound Types [BW98, §6]. Even with little previous experience on mechanical theorem proving, he had remarkably little trouble understanding and modifying the formalization and the proofs for his purpose and thus commends the system for its well-structuredness.

The use of theorem proving system on the one hand causes much work: getting familiar with the system, coping with its limitations or helping to reduce them, and being forced to carry out everything in full detail. On the other hand, both for the formalization and the proofs, machine support was indispensable. Otherwise there would have been plenty of opportunity for omissions and inaccuracies like type errors and inconsistencies, but also the sheer number of definitions to keep in mind and inferences to perform by hand would be overwhelming. This is particularly true within a non-trivial project where many iterations are performed, leading to frequent replay of the large proofs with often subtle, but possibly crucial differences. Due to the number of rules in the operational and axiomatic semantics, in the inductive proofs there are many cases involving a great amount of detail to be considered, for which the partial automation of the theorem prover is of great help.

7.3 Further Work

There are several ways in which this work can — and probably will — be extended and applied.

Extensions of the Model Two important features of the static semantics, name spaces and visibility control, are still missing. It should not be too difficult to add them to the formalization and adapt the proof of type soundness accordingly. This is part of the work to be done for the new European Project VerifiCard [J^+b].

[1] Many of the proofs technically belonging to the theories on the static and dynamic semantics are in fact needed for the proof of type soundness only and thus could be re-assigned to this line.

Extending the model of the dynamic semantics by adding concurrency is much more challenging: to this end one has to resort to the inconvenient transition semantics and it would be wise to prove the equivalence to the evaluation semantics already present. The memory model has to be enhanced, probably along the lines given by Wirsing et al. [CKRW97]. Since the general theory and methodology for verifying concurrent program is still under development, the task of inventing a suitable verification logic for Java, possibly as an extension of our current axiomatic semantics, will be worth at least another PhD thesis.

Modifications of Java Our formalization of current Java is a good starting point for investigating future language extensions. There is already interest in using the model for investigating the type safety of the planned Generic Java [ONO00]. This task has been proposed as a Master's thesis. To our knowledge, there are currently no other plans for official changes to Java, but there is much room for research at least in academia on improving, specializing, and extending the language.

Compiler Verification An extension of our model is going to be used, also within the Project VerifiCard, as the source-level reference semantics for compiler verification. During the project it might turn out that minor parts of the formalization should be adapted in order to facilitate the necessary proofs.

Program Verification Our Hoare logic is not yet fully suited for actual application in program verification. It should be helpful — and not difficult — to identify and deduce simpler specialized versions of many rules more convenient to apply in standard, *e.g.* exception-free, situations. Furthermore, better support by tailored verification tools like an automatic verification condition generator and an some advanced methodology for handling *e.g.* method specifications and object references will surely ease the pain.

Support for Program Design The current stage of the Project Bali [NOPK] includes an application for verifying the implementation of high-level specifications: formalizing the Object Constraint Language of UML also within Isabelle/HOL and tightly connecting it with our axiomatic semantics.

7.4 Final Statement

By the example of treating Java with Isabelle/HOL, we have demonstrated that

machine-checking the design and meta-theory
of realistic programming languages has become feasible.

Such an undertaking is still difficult and costly, yet we believe that for practically important languages it is worth while. May the results of this work, both experience and techniques, be encouraging and helpful for further research and applications in this direction.

Appendix

Isabelle Theory Sources

```
(*  Title:      Isabelle/Bali/Basis.thy
    ID:         $Basis.thy,v 1.29 2000/11/14 09:35:34 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Definitions extending HOL as logical basis of Bali
*)

Basis = PreBasis +

syntax
  "3" :: nat   ("3")
  "4" :: nat   ("4")
translations
  "3" == "Suc 2"
  "4" == "Suc 3"

constdefs
          the_Inl  :: "'a + 'b ⇒ 'a"
         "the_Inl x ≡ εa. x = Inl a"
          the_Inr  :: "'a + 'b ⇒ 'b"
         "the_Inr x ≡ εb. x = Inr b"

datatype ('a, 'b, 'c) sum3 = In1 'a | In2 'b | In3 'c
constdefs
          the_In1  :: "('a, 'b, 'c) sum3 ⇒ 'a"
         "the_In1 x ≡ εa. x = In1 a"
          the_In2  :: "('a, 'b, 'c) sum3 ⇒ 'b"
         "the_In2 x ≡ εb. x = In2 b"
          the_In3  :: "('a, 'b, 'c) sum3 ⇒ 'c"
         "the_In3 x ≡ εc. x = In3 c"

syntax
         In1l   :: "'al ⇒ ('al + 'ar, 'b, 'c) sum3"
         In1r   :: "'ar ⇒ ('al + 'ar, 'b, 'c) sum3"
translations
        "In1l e" == "In1 (Inl e)"
        "In1r c" == "In1 (Inr c)"

translations
  "option"<= (type) "Option.option"
  "list"  <= (type) "List.list"
  "sum3"  <= (type) "Basis.sum3"

syntax
  fun_sum :: "('a => 'c) => ('b => 'c) => (('a+'b) => 'c)" (infixr "'(+')"80)
translations
 "fun_sum" == "sum_case"
```

```
syntax
  "@Oall" :: [pttrn, 'a option, bool] => bool   ("(3! _:_:/ _)" [0,0,10] 10)
  "@Oex"  :: [pttrn, 'a option, bool] => bool   ("(3? _:_:/ _)" [0,0,10] 10)

syntax (symbols)
  "@Oall" :: [pttrn, 'a option, bool] => bool   ("(3∀_∈_:/ _)"  [0,0,10] 10)
  "@Oex"  :: [pttrn, 'a option, bool] => bool   ("(3∃_∈_:/ _)"  [0,0,10] 10)

translations
  "! x:A: P"    == "! x:o2s A. P"
  "? x:A: P"    == "? x:o2s A. P"

constdefs
  unique   :: "('a × 'b) list ⇒ bool"
 "unique ≡ nodups ∘ map fst"

consts
  lsplit         :: "[['a, 'a list] => 'b, 'a list] => 'b"
defs
  lsplit_def     "lsplit == %f l. f (hd l) (tl l)"
(*  list patterns -- extends pre-defined type "pttrn" used in abstractions *)
syntax
  "_lpttrn"    :: [pttrn,pttrn] => pttrn     ("_#/_" [901,900] 900)
translations
  "%y#x#xs. b"  == "lsplit (%y x#xs. b)"
  "%x#xs  . b"  == "lsplit (%x xs  . b)"

syntax

  "@dummy_pat"   :: pttrn    ("'_")

end

ML
fun dummy_pat_tr [] = Free ("_",dummyT)
  | dummy_pat_tr ts = raise TERM ("dummy_pat_tr", ts);

val parse_translation = ("@dummy_pat", dummy_pat_tr)::parse_translation;
```

```
(*  Title:      Isabelle/Bali/Table.thy
    ID:         $Table.thy,v 1.29 2000/11/27 15:20:15 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Abstract tables and their implementation as lists

design issues:
* definition of table: infinite map vs. list vs. finite set
  list chosen, because:
  +  a priori finite
  +  lookup is more operational than for finite set
  -  not very abstract, but function table converts it to abstract mapping
* coding of lookup result: Some/None vs. value/arbitrary
   Some/None chosen, because:
  ++ makes definedness check possible (applies also to finite set),
     which is important for the type standard, hiding/overriding, etc.
     (though it may perhaps be possible at least for the operational semantics
      to treat programs as infinite, i.e. where classes, fields, methods etc.
      of any name are considered to be defined)
  -  sometimes awkward case distinctions, alleviated by operator 'the'
*)


Table = Basis +

types ('a, 'b) table    (* table with key type 'a and contents type 'b *)
      = "'a ⇀ 'b"
      ('a, 'b) tables   (* non-unique table with key 'a and contents 'b *)
      = "'a ⇒ 'b set"

syntax
  table_of      :: "('a × 'b) list ⇒ ('a, 'b) table"    (* concrete table *)

translations
  "table_of" == "map_of"

  (type)"'a ⇀ 'b"       <= (type)"'a ⇒ 'b Option.option"
  (type)"('a, 'b) table" <= (type)"'a ⇀ 'b"

consts
  Un_tables      :: "('a, 'b) tables set ⇒ ('a, 'b) tables"
  overrides      :: "('a, 'b) tables     ⇒ ('a, 'b) tables ⇒
                     ('a, 'b) tables"             (infixl "⊕⊕" 100)
  hidings_entails:: "('a, 'b) tables ⇒ ('a, 'c) tables ⇒
                     ('b ⇒ 'c ⇒ bool) ⇒ bool"   ("_ hidings _ entails _" 20)
  (* variant for unique table: *)
  hiding_entails :: "('a, 'b) table  ⇒ ('a, 'c) table  ⇒
                     ('b ⇒ 'c ⇒ bool) ⇒ bool"   ("_ hiding _ entails _"  20)
```

```
defs
  Un_tables_def       "Un_tables ts   ≡ λk. ⋃t∈ts. t k"
  overrides_def       "s ⊕⊕ t         ≡ λk. if t k = {} then s k else t k"
  hidings_entails_def "t hidings s entails R ≡ ∀k. ∀x∈t k. ∀y∈s k. R x y"
  hiding_entails_def  "t hiding  s entails R ≡ ∀k. ∀x∈t k: ∀y∈s k: R x y"

consts
  atleast_free :: "('a ~=> 'b) => nat => bool"

primrec
 "atleast_free m 0       = True"
 "atleast_free m (Suc n) = (? a. m a = None & (!b. atleast_free (m(a|->b)) n))"

end
```

```
(*  Title:      Isabelle/Bali/Name.thy
    ID:         $Name.thy,v 1.5 2000/06/29 19:35:31 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java names

simplifications:
no packages, thus no internal structure of names
*)

Name = Basis +

(* cf. 6.5 *)
types   tnam            (* ordinary type name, i.e. class or interface name *)
        ename           (* expression name, i.e. variable or field name *)
        mname           (* method name *)
arities tnam, ename, mname :: term

types    lname           (* names for local variables and the This pointer *)
        = "ename + unit"
syntax   EName, This :: lname
translations
  "lname" <= (type) "ename + unit"
  "EName" => "Inl"
  "This"  => "Inr ()"

datatype xname          (* names of standard exceptions *)
        = Throwable
        | NullPointer | OutOfMemory | ClassCast
        | NegArrSize  | IndOutBound | ArrStore

datatype tname          (* type names for standard classes and other type names *)
        = Object
        | SXcpt   xname
        | TName   tnam

translations
  "mname" <= (type) "Name.mname"
  "xname" <= (type) "Name.xname"
  "tname" <= (type) "Name.tname"
  "ename" <= (type) "Name.ename"

end
```

```
(*  Title:      Isabelle/Bali/Type.thy
    ID:         $Type.thy,v 1.19 2000/05/02 09:40:54 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java types

simplifications:
* only the most important primitive types
* the null type is regarded as reference type
*)
Type = Name +

datatype prim_ty        (* primitive type, cf. 4.2 *)
        = Void          (* 'result type' of void methods *)
        | Boolean
        | Integer


datatype ref_ty         (* reference type, cf. 4.3 *)
        = NullT         (* null type, cf. 4.1 *)
        | IfaceT tname  (* interface type *)
        | ClassT tname  (* class type *)
        | ArrayT ty     (* array type *)

and ty                  (* any type, cf. 4.1 *)
        = PrimT prim_ty (* primitive type *)
        | RefT  ref_ty  (* reference type *)

translations
  "prim_ty" <= (type) "Type.prim_ty"
  "ref_ty"  <= (type) "Type.ref_ty"
  "ty"      <= (type) "Type.ty"

syntax
         NT     :: "          ty"
         Iface  :: "tname  ⇒ ty"
         Class  :: "tname  ⇒ ty"
         Array  :: "ty     ⇒ ty"        ("_.[]" [90] 90)

translations
        "NT"      == "RefT   NullT"
        "Iface I" == "RefT (IfaceT I)"
        "Class C" == "RefT (ClassT C)"
        "T.[]"    == "RefT (ArrayT T)"

constdefs
  the_Class :: "ty ⇒ tname"
 "the_Class T ≡ εC. T = Class C"
end
```

```
(*  Title:      Isabelle/Bali/Value.thy
    ID:         $Value.thy,v 1.3 2000/07/14 14:48:59 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java values
*)

Value = Type +

types   loc             (* locations, i.e. abstract references on objects *)
arities loc :: term

datatype val_ (** name not 'val' because of nasty clash with ML token 'val' **)
        = Unit          (* dummy result value of void methods *)
        | Bool bool     (* Boolean value *)
        | Intg int      (* integer value *)
        | Null          (* null reference *)
        | Addr loc      (* addresses, i.e. locations of objects *)
types         val   =         val_
translations "val" <= (type) "val_"
             "val" <= (type) "Term.val_"
             "loc" <= (type) "Term.loc"

constdefs
  the_Bool   :: "val ⇒ bool"  "the_Bool v ≡  εb. v = Bool b"
  the_Intg   :: "val ⇒ int"   "the_Intg v ≡  εi. v = Intg i"
  the_Addr   :: "val ⇒ loc"   "the_Addr v ≡  εa. v = Addr a"

types   dyn_ty      = "loc ⇒ ty option"
consts
  typeof        :: "dyn_ty ⇒ val ⇒ ty option"
  defpval       :: "prim_ty ⇒ val"      (* default value for primitive types *)
  default_val   :: "     ty ⇒ val"      (* default value for all types *)

primrec "typeof dt  Unit    = Some (PrimT Void)"
        "typeof dt (Bool b) = Some (PrimT Boolean)"
        "typeof dt (Intg i) = Some (PrimT Integer)"
        "typeof dt  Null    = Some NT"
        "typeof dt (Addr a) = dt a"

primrec "defpval Void    = Unit"
        "defpval Boolean = Bool False"
        "defpval Integer = Intg #0"
primrec "default_val (PrimT pt) = defpval pt"
        "default_val (RefT  r ) = Null"

end
```

```
(*  Title:      Isabelle/Bali/Term.thy
    ID:         $Term.thy,v 1.43 2000/11/21 07:50:57 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Java expressions and statements

design issues:
* invocation frames for local variables could be reduced to special static
  objects (one per method). This would reduce redundancy, but yield a rather
  non-standard execution model more difficult to understand.
* method bodies separated from calls to handle assumptions in axiomat. semantics
  NB: Body is intended to be in the environment of the _called_ method.
* class initialization is regarded as (auxiliary) statement (required for AxSem)
simplifications:
* expression statement allowed for any expression
* no unary, binary, etc, operators
* This  is modeled as a special non-assignable local variable
* Super is modeled as a general expression with the same value as This
* access to field x in current class via This.x
* NewA creates only one-dimensional arrays;
  initialization of further subarrays may be simulated with nested NewAs
* The 'Lit' constructor is allowed to contain a reference value.
  But this is assumed to be prohibited in the input language, which is enforced
  by the type-checking rules.
* a call of a static method via a type name may be simulated by a dummy variable
* result expression instead of return statement (see Decl.thy)
* no nested blocks with inner local variables
* no synchronized statements
* no secondary forms of if, while (e.g. no for) (may be easily simulated)
* no switch, break, continue, no labels (may be simulated with while)
* the try_catch_finally statement is divided into the try_catch statement and
  a finally statement, which may be considered as try..finally with empty catch
* the try_catch statement has exactly one catch clause; multiple ones can be
  simulated with instanceof
* the compiler is supposed to add the annotations {_} during type-checking. This
  transformation is left out as its result is checked by the type rules anyway
*)
Term = Value +

datatype inv_mode                  (* invocation mode for method calls *)
        = Static                   (* static *)
        | SuperM                   (* super  *)
        | IntVir                   (* interface or virtual *)
types    sig             (* signature of a method, cf. 8.4.2  *)
        = "mname × ty list"        (* acutally belongs to Decl.thy *)
datatype var
        = LVar                  lname(* local variable (incl. parameters) *)
        | FVar  tname bool expr ename(*class field*)("{_,_}_.._"[10,10,85,99]90)
        | AVar        expr expr      (* array component *) ("_.[_]"[90,10   ]90)
```

```
and expr
        = NewC tname              (* class instance creation *)
        | NewA ty expr            (* array creation *) ("New _[_]"[99,10   ]85)
        | Cast ty expr            (* type cast  *)
        | Inst expr ref_ty       (* instanceof *)     ("_ InstOf _"[85,99] 85)
        | Lit  val                (* literal value, references not allowed *)
        | Super                   (* special Super keyword *)
        | Acc  var                (* variable access *)
        | Ass  var expr           (* variable assign *) ("_:=_"   [90,85   ]85)
        | Cond expr expr expr     (* conditional *)  ("_ ? _ : _" [85,85,80]80)
        | Call ref_ty ref_ty inv_mode expr mname (* method call *)
          (ty list) (expr list) ("{_,_,_}_.._'( {_}_')"[10,10,10,85,99,10,10]85)
        | Methd tname sig         (*   (folded) method *)
        | Body tname stmt expr    (* (unfolded) method body *)
and  stmt
        = Skip                    (* empty      statement *)
        | Expr  expr              (* expression statement *)
        | Comp  stmt stmt         ("_;; _"                     [      66,65]65)
        | If_   expr stmt stmt   ("If'(_') _ Else _"           [   80,79,79]70)
        | Loop  expr stmt         ("While'(_') _"              [      80,79]70)
        | Throw expr
        | TryC  stmt
                tname ename stmt  ("Try _ Catch'(_ _') _"       [79,99,80,79]70)
        | Fin   stmt stmt         ("_ Finally _"               [      79,79]70)
        | init  tname             (* class initialization *)

types term = "(expr+stmt, var, expr list) sum3"
translations
  "sig"   <= (type) "mname × ty list"
  "var"   <= (type) "Term.var"
  "expr"  <= (type) "Term.expr"
  "stmt"  <= (type) "Term.stmt"
  "term"  <= (type) "(expr+stmt, var, expr list) sum3"

syntax
  this    :: expr
  LAcc    :: "ename ⇒         expr" ("!!")
  LAss    :: "ename ⇒ expr ⇒ stmt" ("_:==_" [90,85] 85)
  StatRef :: "ref_ty ⇒ expr"
translations
 "this"       == "Acc (LVar This)"
 "!!v"        == "Acc (LVar (Inl v))"
 "v:==e"      == "Expr (Ass (LVar (Inl v)) e)"
 "StatRef rt" == "Cast (RefT rt) (Lit Null)"

constdefs
  is_stmt :: "term ⇒ bool"
 "is_stmt t ≡ ∃c. t=In1r c"
end
```

```
(*  Title:      Isabelle/Bali/Decl.thy
    ID:         $Decl.thy,v 1.40 2000/11/27 15:20:15 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Field, method, interface, and class declarations, whole Java programs

simplifications:
* the only field and method modifier is static
* no constructors, which may be simulated by new + suitable methods
* there is just one global initializer per class, which can simulate all others

* no throws clause
* result statement replaced by result expression (evaluated at the end of the
  execution of the body; transformation is always possible (with goto, while)
* a void method is replaced by one that returns Unit (of dummy type Void)

* no interface modifiers yet, i.e. every interface is public
* no interface fields

* no class modifiers yet, i.e. every class is public, non-final
* every class has an explicit superclass (unused for Object)
* the (standard) methods of Object and of standard exceptions are not specified

* no packages
* no main method
*)

Decl = Term + Table + (** order is significant, because of clash for "var" **)

types   modi = bool     (* modifier: static *)
        field = "modi × ty"
        fdecl = "ename × field" (* field declaration, cf. 8.3 *)

translations
  "field" <= (type) "bool × ty"
  "fdecl" <= (type) "ename × field"

types (*sig: see Term.thy *)

        mhead           (* method head (excluding signature) *)
        = "modi × ename list × ty"
        (* modifier, parameter names, result type *)

        mbody           (* method body *)
        = "(ename × ty) list × stmt × expr"
        (* local variables, block+result expression *)

        methd           (* method in a class *)
        = "mhead × mbody"
```

```
        mdecl           (* method declaration in a class *)
        = "sig × methd"

translations
  "mhead" <= (type) "bool × ename list × ty"
  "mbody" <= (type) "(ename × ty) list × stmt × expr"
  "methd" <= (type) "mhead × mbody"
  "mdecl" <= (type) "sig × methd"

syntax
  static    :: "modi  ⇒ bool"
  mrt       :: "mhead ⇒ ty"

translations
        "static"       => "id"
        "mrt mh"       => "snd (snd mh)"


types   ibody           (* interface body *)
        = "(sig × mhead) list"
        (* methods *)

        iface           (* interface *)
        = "tname list × ibody"
        (* superinterface list *)

        idecl           (* interface declaration, cf. 9.1 *)
        = "tname × iface"

translations
  "ibody" <= (type) "(sig × mhead) list"
  "iface" <= (type) "tname list × ibody"
  "idecl" <= (type) "tname × iface"


types   cbody           (* class body *)
        = "fdecl list × mdecl list × stmt"
        (* fields, methods, initializer *)

        class           (* class *)
        = "tname × tname list × cbody"
        (* superclass, implemented interfaces *)

        cdecl           (* class declaration, cf. 8.1 *)
        = "tname × class"

translations
  "cbody" <= (type) "fdecl list × mdecl list × stmt"
  "class" <= (type) "tname × tname list × cbody"
  "cdecl" <= (type) "tname × class"
```

```
consts

  Object_mdecls  :: "mdecl list" (* methods of Object *)
  SXcpt_mdecls   :: "mdecl list" (* methods of SXcpts *)
  ObjectC ::        "cdecl"      (* declaration  of root      class   *)
  SXcptC  ::"xname ⇒ cdecl"      (* declarations of throwable classes *)
  standard_classes :: cdecl list

defs

ObjectC_def "ObjectC  ≡ (Object  , (arbitrary      ,[],[],Object_mdecls,Skip))"
SXcptC_def  "SXcptC xn≡ (SXcpt xn, (if xn = Throwable then Object else
                                     SXcpt Throwable,[],[],SXcpt_mdecls,Skip))"
standard_classes_def "standard_classes ≡ [ObjectC, SXcptC Throwable,
                SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
                SXcptC NegArrSize , SXcptC IndOutBound, SXcptC ArrStore]"


(* programs *)
types prog  =        "idecl list × cdecl list"
translations
     "prog"<= (type) "idecl list × cdecl list"

syntax
  iface     :: "prog  ⇒ (tname, iface) table"
  class     :: "prog  ⇒ (tname, class) table"
  is_iface  :: "prog  ⇒ tname  ⇒ bool"
  is_class  :: "prog  ⇒ tname  ⇒ bool"

translations
           "iface G I" == "table_of (fst G) I"
           "class G C" == "table_of (snd G) C"
        "is_iface G I" == "iface G I ≠ None"
        "is_class G C" == "class G C ≠ None"

consts
  is_type :: "prog ⇒     ty ⇒ bool"
  isrtype :: "prog ⇒ ref_ty ⇒ bool"

primrec "is_type G (PrimT pt)  = True"
        "is_type G (RefT  rt)  = isrtype G rt"
        "isrtype G (NullT    ) = True"
        "isrtype G (IfaceT tn) = is_iface G tn"
        "isrtype G (ClassT tn) = is_class G tn"
        "isrtype G (ArrayT T ) = is_type  G T"
```

```
(* subinterface and subclass relation, in anticipation of TypeRel.thy *)
consts
  subint1,
  subcls1  :: "prog ⇒ (tname × tname) set"
defs
  subint1_def   "subint1 G ≡ {(I,J). ∃i∈iface G I: J∈set (fst i)}"
  subcls1_def   "subcls1 G ≡ {(C,D). C≠Object ∧ (∃c∈class G C: fst c = D)}"

(* well-structured programs *)
constdefs
  ws_idecl :: "prog ⇒ tname ⇒ tname list ⇒ bool"
 "ws_idecl G I si ≡ ∀J∈set si.  is_iface G J   ∧ (J,I)∉(subint1 G)^+"

  ws_cdecl :: "prog ⇒ tname ⇒ tname ⇒ bool"
 "ws_cdecl G C sc ≡ C≠Object ⟶ is_class G sc ∧ (sc,C)∉(subcls1 G)^+"

  ws_prog  :: "prog ⇒ bool"
 "ws_prog G ≡ (∀(I,(si,ib))∈set (fst G). ws_idecl G I si) ∧
              (∀(C,(sc,cb))∈set (snd G). ws_cdecl G C sc)"

(* auxiliary well-founded relation for the recursion operators below *)
constdefs
  ws_wfrel :: "(prog ⇒ (tname × tname) set) ⇒
               ((prog × tname) × (prog × tname)) set"
 "ws_wfrel R ≡ {((G,T),(G',T')). G' = G ∧ ws_prog G ∧ (T',T) ∈ R G}"

(* general operators for recursion over the interface and class hiearchies *)
consts
  iface_rec   :: "prog × tname ⇒       (tname ⇒ ibody ⇒ 'a set ⇒ 'a) ⇒ 'a"
  class_rec   :: "prog × tname ⇒ 'a ⇒ (tname ⇒ cbody ⇒ 'a     ⇒ 'a) ⇒ 'a"

recdef iface_rec "ws_wfrel subint1" congs image_cong
"iface_rec (G,I) = (λf. case iface G I of None ⇒ arbitrary | Some (si,ib) ⇒
                    if ws_prog G then f I ib ((λJ. iface_rec (G,J) f)``set si)
                    else arbitrary)"

recdef class_rec "ws_wfrel subcls1"
"class_rec(G,C) = (λt f. case class G C of None⇒ arbitrary | Some (sc,si,cb)⇒
       if ws_prog G then f C cb (if C = Object then t else class_rec (G,sc) t f)
       else arbitrary)"

types
  fspec = "ename × tname"

consts
  imethds       :: "prog ⇒ tname ⇒ ( sig   , tname  ×  mhead) tables"
  cmethd        :: "prog ⇒ tname ⇒ ( sig   , tname  ×  methd) table"
  fields        :: "prog ⇒ tname ⇒ ((ename × tname) ×  field) list"
  cfield        :: "prog ⇒ tname ⇒ ( ename , tname  ×  field) table"
```

```
defs
  (* methods of an interface, with overriding and inheritance, cf. 9.2 *)
  imeths_def "imethds G I ≡ iface_rec (G,I)       (λI     ms      ts.
       (Un_tables ts) ⊕⊕ (o2s ∘ table_of (map (λ(s,m). (s,I,m)) ms)))"

  (* methods of a class, with inheritance, overriding and hiding, cf. 8.4.6 *)
  cmethd_def "cmethd  G C ≡ class_rec (G,C) empty (λC (fs,ms,ini) ts.
                           ts ++ table_of (map (λ(s,m). (s,C,m)) ms))"

  (* list of fields of a class, including inherited and hidden ones *)
  fields_def "fields G C ≡ class_rec (G,C) []    (λC (fs,ms,ini) ts.
                                             map (λ(n,t). ((n,C),t)) fs @ ts)"

  (* fields of a class, with inheritance and hiding, cf. 8.3 *)
  cfield_def "cfield G C ≡ table_of((map (λ((n,d),T).(n,(d,T)))) (fields G C))"


constdefs
  is_methd :: "prog ⇒ tname ⇒ sig ⇒ bool"
 "is_methd G ≡ λC sig. is_class G C ∧ cmethd G C sig ≠ None"

end
```

```
(*  Title:      Isabelle/Bali/TypeRel.thy
    ID:         $TypeRel.thy,v 1.31 2000/07/25 21:54:10 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

The relations between Java types

simplifications:
* subinterface, subclass and widening relation includes identity

improvements over Java Specification 1.0:
* narrowing reference conversion also in cases where the return types of a pair
  of methods common to both types are in widening (rather identity) relation
* one could add similar constraints also for other cases

design issues:
* the type relations do not require is_type for their arguments
* the subint1 and subcls1 relations imply is_iface/is_class for their first
  arguments, which is required for their finiteness

*)

TypeRel = Decl +

consts

(*subint1, in Decl.thy*)                     (* direct subinterface   *)
(*subint,  by translation*)                  (*        subinterface   *)
(*subcls1, in Decl.thy*)                     (* direct subclass       *)
(*subcls,  by translation*)                  (*        subclass       *)
  implmt1,                                   (* direct implementation *)
  implmt   :: "prog ⇒ (tname × tname) set" (*        implementation *)
  widen,                                     (*        widening       *)
  narrow,                                    (*        narrowing      *)
  cast     :: "prog ⇒ (ty    × ty   ) set" (*        casting        *)

syntax

 "@subint1" :: "prog => [tname, tname] => bool" ("_|-_<:I1_" [71,71,71] 70)
 "@subint"  :: "prog => [tname, tname] => bool" ("_|-_<=:I _"[71,71,71] 70)
 "@subcls1" :: "prog => [tname, tname] => bool" ("_|-_<:C1_" [71,71,71] 70)
 "@subcls"  :: "prog => [tname, tname] => bool" ("_|-_<=:C _"[71,71,71] 70)
 "@implmt1" :: "prog => [tname, tname] => bool" ("_|-_~>1_"  [71,71,71] 70)
 "@implmt"  :: "prog => [tname, tname] => bool" ("_|-_~>_"   [71,71,71] 70)
 "@widen"   :: "prog => [ty   , ty   ] => bool" ("_|-_<=:_"  [71,71,71] 70)
 "@narrow"  :: "prog => [ty   , ty   ] => bool" ("_|-_:>_"   [71,71,71] 70)
 "@cast"    :: "prog => [ty   , ty   ] => bool" ("_|-_<=:? _"[71,71,71] 70)
```

```
syntax (symbols)

  "@subint1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≺I1_"  [71,71,71] 70)
  "@subint"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≼I _"  [71,71,71] 70)
  "@subcls1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≺C1_"  [71,71,71] 70)
  "@subcls"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_≼C _"  [71,71,71] 70)
  "@implmt1" :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_⇝1_"   [71,71,71] 70)
  "@implmt"  :: "prog ⇒ [tname, tname] ⇒ bool" ("_⊢_⇝_"    [71,71,71] 70)
  "@widen"   :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≼_"    [71,71,71] 70)
  "@narrow"  :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≻_"    [71,71,71] 70)
  "@cast"    :: "prog ⇒ [ty   , ty   ] ⇒ bool" ("_⊢_≼? _"  [71,71,71] 70)

translations

        "G⊢I ≺I1 J" == "(I,J) ∈ subint1 G"
        "G⊢I ≼I  J" == "(I,J) ∈(subint1 G)^*" (* cf. 9.1.3 *)
        "G⊢C ≺C1 D" == "(C,D) ∈ subcls1 G"
        "G⊢C ≼C  D" == "(C,D) ∈(subcls1 G)^*" (* cf. 8.1.3 *)
        "G⊢C ⇝1 I" == "(C,I) ∈ implmt1 G"
        "G⊢C ⇝  I" == "(C,I) ∈ implmt  G"
        "G⊢S ≼   T" == "(S,T) ∈ widen   G"
        "G⊢S ≻   T" == "(S,T) ∈ narrow  G"
        "G⊢S ≼?  T" == "(S,T) ∈ cast    G"


defs

  (* direct subinterface in Decl.thy, cf. 9.1.3 *)
  (* direct subclass     in Decl.thy, cf. 8.1.3 *)

  (* direct implementation, cf. 8.1.3 *)
  implmt1_def "implmt1 G≡{(C,I). C≠Object ∧ (∃c∈class G C: I∈set (fst(snd c)))}"

inductive "implmt G" intrs                    (* cf. 8.1.4 *)

  direct         "G⊢C⇝1J               ⟹ G⊢C⇝J"
  subint        "⟦G⊢C⇝1I; G⊢I≼I J⟧  ⟹ G⊢C⇝J"
  subcls1       "⟦G⊢C≺C1D; G⊢D⇝J ⟧  ⟹ G⊢C⇝J"

inductive "widen G" intrs (*widening, viz. method invocation conversion, cf. 5.3
                            i.e. kind of syntactic subtyping *)
  refl                      "G⊢        T≼T" (*identity conversion, cf. 5.1.1 *)
  subint   "G⊢I≼I J          ⟹ G⊢  Iface I≼ Iface J"(*wid.ref.conv., cf. 5.1.4 *)
  int_obj                  "G⊢  Iface I≼ Class Object"
  subcls   "G⊢C≼C D           ⟹ G⊢  Class C≼ Class D"
  implmt   "G⊢C⇝I             ⟹ G⊢  Class C≼ Iface I"
  null                     "G⊢       NT≼ RefT R"
  arr_obj                  "G⊢     T.[]≼ Class Object"
  array    "G⊢RefT S≼RefT T ⟹ G⊢RefT S.[]≼ RefT T.[]"
```

```
(* all properties of narrowing and casting conversions we actually need *)
(* these can easily be proven from the definitions below *)
(*
rules
  cast_RefT2   "G⊢S≼? RefT R   ⟹ ∃t. S=RefT t"
  cast_PrimT2  "G⊢S≼? PrimT pt ⟹ ∃t. S=PrimT t ∧ G⊢PrimT t≼PrimT pt"
*)

constdefs
  widens :: "prog ⇒ [ty list, ty list] ⇒ bool" ("_⊢_[≼]_" [71,71,71] 70)
 "G⊢Ts[≼]Ts' ≡ list_all2 (λT T'. G⊢T≼T') Ts Ts'"

(* more detailed than necessary for type-safety, see above rules. *)
inductive "narrow G" intrs (* narrowing reference conversion, cf. 5.1.5 *)

  subcls "G⊢C≼C D               ⟹ G⊢     Class D≻Class C"
  implmt "¬G⊢C↝I                ⟹ G⊢     Class C≻Iface I"
  obj_arr                  "G⊢Class Object≻T.[]"
  int_cls                  "G⊢     Iface I≻Class C"
  subint "imethds G I hidings imethds G J entails
         (λ(md, mh   ) (md',mh').  G⊢mrt mh≼mrt mh') ⟹
         ¬G⊢I≼I J               ⟹ G⊢     Iface I≻Iface J"
  array  "G⊢RefT S≻RefT T     ⟹ G⊢   RefT S.[]≻RefT T.[]"


inductive "cast G" intrs (* casting conversion, cf. 5.5 *)

  widen   "G⊢S≼T ⟹ G⊢S≼? T"
  narrow  "G⊢S≻T ⟹ G⊢S≼? T"

end
```

```
(*  Title:      Isabelle/Bali/WellType.thy
    ID:         $WellType.thy,v 1.42 2000/07/11 20:28:42 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Well-typedness of Java programs

improvements over Java Specification 1.0:
* methods of Object can be called upon references of interface or array type

simplifications:
* the type rules include all static checks on statements and expressions, e.g.
  definedness of names (of parameters, locals, fields, methods)

design issues:
* unified type judgment for statements, variables, expressions, expression lists
* statements are typed like expressions with dummy type Void
* the typing rules take an extra argument that is capable of determining
  the dynamic type of objects. Therefore, they can be used for both
  checking static types and determining runtime types in transition semantics.
*)
WellType = TypeRel +

types   lenv
        = "(lname, ty) table"   (* local variables, including This *)

        env
        = "prog × lenv"         (* program, locals *)

syntax
  prg     :: "env ⇒ prog"
  lcl     :: "env ⇒ lenv"
translations
  "lenv" <= (type) "(lname, ty) table"
   "env" <= (type) "prog × lenv"
  "prg"   => "fst"
  "lcl"   => "snd"

types
  emhead = "ref_ty × mhead"

consts
 cmheads :: "prog ⇒ tname  ⇒ sig ⇒ emhead set"
  mheads :: "prog ⇒ ref_ty ⇒ sig ⇒ emhead set"

defs
 cmheads_def
"cmheads G C ≡ λsig. (λ(C,(h,b)). (ClassT C,h)) `` o2s (cmethd G C sig)"
```

```
primrec

"mheads G   NullT    = (λsig. {})"
"mheads G (IfaceT I) = (λsig. (λ(I,h).(IfaceT I,h)) ‘‘ imethds G I sig ∪
                       cmheads G Object sig)"
"mheads G (ClassT C) = cmheads G C"
"mheads G (ArrayT T) = cmheads G Object"
(* more detailed than necessary for type-safety, see below. *)

constdefs

  (* applicable methods, cf. 15.11.2.1 *)
  appl_methds   :: "prog ⇒  ref_ty ⇒ sig ⇒ (emhead × ty list)   set"
 "appl_methds G rt ≡ λ(mn, pTs). {(mh,pTs') |mh pTs'.
                                   mh ∈ mheads G rt (mn, pTs') ∧ G⊢pTs[≼]pTs'}"

  (* more specific methods, cf. 15.11.2.2 *)
  more_spec     :: "prog ⇒ emhead × ty list ⇒ emhead × ty list ⇒ bool"
 "more_spec G ≡ λ(mh,pTs). λ(mh',pTs'). G⊢pTs[≼]pTs'"
(*more_spec G ≡λ((d,h),pTs). λ((d',h'),pTs'). G⊢RefT d≼RefT d'∧G⊢pTs[≼]pTs'*)

  (* maximally specific methods, cf. 15.11.2.2 *)
   max_spec     :: "prog ⇒  ref_ty ⇒ sig ⇒ (emhead × ty list)   set"

 " max_spec G rt sig ≡{m. m ∈appl_methds G rt sig ∧
                        (∀m'∈appl_methds G rt sig. more_spec G m' m ⟶ m'=m)}"

(*
rules (* all properties of more_spec, appl_methods and max_spec we actually need
         these can easily be proven from the above definitions @*)

max_spec2mheads "max_spec G rt (mn, pTs) = insert (mh, pTs') A ⟹
      mh∈mheads G rt (mn, pTs') ∧ G⊢pTs[≼]pTs'"
*)

constdefs

  empty_dt :: "dyn_ty"
 "empty_dt ≡ λa. None"

  invmode :: "modi ⇒ expr ⇒ inv_mode"
 "invmode m e ≡ if static m then Static else if e=Super then SuperM else IntVir"

types tys  =        "ty + ty list"
translations
  "tys"   <= (type) "ty + ty list"
```

```
consts
  wt    :: "(env   × dyn_ty   ×  term × tys) set"
(*wt    :: " env ⇒ dyn_ty ⇒ (term × tys) set"@ not feasible because of
                                                 changing env in Try stmt *)
syntax

wt      :: "env ⇒ dyn_ty ⇒ [term,tys] ⇒ bool"  ("_,_|=_::_" [51,51,51,51] 50)
wt_stmt :: "env ⇒ dyn_ty ⇒  stmt       ⇒ bool" ("_,_|=_:<>" [51,51,51   ] 50)
ty_expr :: "env ⇒ dyn_ty ⇒ [expr ,ty ] ⇒ bool" ("_,_|=_:-_" [51,51,51,51] 50)
ty_var  :: "env ⇒ dyn_ty ⇒ [var  ,ty ] ⇒ bool" ("_,_|=_:=_" [51,51,51,51] 50)
ty_exprs:: "env ⇒ dyn_ty ⇒ [expr list,
                             ty   list] ⇒ bool" ("_,_|=_:#_" [51,51,51,51] 50)
syntax (xsymbols)

wt      :: "env ⇒ dyn_ty ⇒ [term, tys] ⇒ bool" ("_,_⊨_::_"  [51,51,51,51] 50)
wt_stmt :: "env ⇒ dyn_ty ⇒  stmt       ⇒ bool" ("_,_⊨_::√"  [51,51,51   ] 50)
ty_expr :: "env ⇒ dyn_ty ⇒ [expr ,ty ] ⇒ bool" ("_,_⊨_::-_" [51,51,51,51] 50)
ty_var  :: "env ⇒ dyn_ty ⇒ [var  ,ty ] ⇒ bool" ("_,_⊨_::=_" [51,51,51,51] 50)
ty_exprs:: "env ⇒ dyn_ty ⇒ [expr list,
                             ty   list] ⇒ bool" ("_,_⊨_::≐_" [51,51,51,51] 50)
translations
        "E,dt⊨t::T" == "(E,dt,t,T) ∈ wt"
        "E,dt⊨s::√"  == "E,dt⊨In1r s::Inl (PrimT Void)"
        "E,dt⊨e::-T" == "E,dt⊨In1l e::Inl T"
        "E,dt⊨e::=T" == "E,dt⊨In2  e::Inl T"
        "E,dt⊨e::≐T" == "E,dt⊨In3  e::Inr T"

syntax (* for purely static typing *)

  wt_      :: "env ⇒ [term, tys] ⇒ bool" ("_|-_::_" [51,51,51] 50)
  wt_stmt_ :: "env ⇒  stmt       ⇒ bool" ("_|-_:<>" [51,51   ] 50)
  ty_expr_ :: "env ⇒ [expr ,ty ] ⇒ bool" ("_|-_:-_" [51,51,51] 50)
  ty_var_  :: "env ⇒ [var  ,ty ] ⇒ bool" ("_|-_:=_" [51,51,51] 50)
  ty_exprs_:: "env ⇒ [expr list,
                       ty   list] ⇒ bool" ("_|-_:#_" [51,51,51] 50)
syntax (xsymbols)

  wt_      :: "env ⇒ [term,tys] ⇒ bool" ("_⊢_::_"  [51,51,51] 50)
  wt_stmt_::  "env ⇒  stmt       ⇒ bool" ("_⊢_::√"  [51,51   ] 50)
  ty_expr_ :: "env ⇒ [expr ,ty ] ⇒ bool" ("_⊢_::-_" [51,51,51] 50)
  ty_var_  :: "env ⇒ [var  ,ty ] ⇒ bool" ("_⊢_::=_" [51,51,51] 50)
  ty_exprs_:: "env ⇒ [expr list,
                       ty   list] ⇒ bool" ("_⊢_::≐_" [51,51,51] 50)

translations
        "E⊢t:: T" == "E,empty_dt⊨t:: T"
        "E⊢s::√" == "E,empty_dt⊨s::√"
        "E⊢e::-T" == "E,empty_dt⊨e::-T"
        "E⊢e::=T" == "E,empty_dt⊨e::=T"
        "E⊢e::≐T" == "E,empty_dt⊨e::≐T"
```

```
inductive wt intrs

(* well-typed statements *)

  Skip                                  "E,dt⊨Skip::√"

  Expr  "⟦E,dt⊨e::-T⟧ ⟹
                                         E,dt⊨Expr e::√"

  Comp  "⟦E,dt⊨c1::√;
          E,dt⊨c2::√⟧ ⟹
                                         E,dt⊨c1;; c2::√"

  (* cf. 14.8 *)
  If    "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c1::√;
          E,dt⊨c2::√⟧ ⟹
                                         E,dt⊨If(e) c1 Else c2::√"

  (* cf. 14.10 *)
  Loop  "⟦E,dt⊨e::-PrimT Boolean;
          E,dt⊨c::√⟧ ⟹
                                         E,dt⊨While(e) c::√"

  (* cf. 14.16 *)
  Throw "⟦E,dt⊨e::-Class tn;
          prg E⊢tn≼C SXcpt Throwable⟧ ⟹
                                         E,dt⊨Throw e::√"
  (* cf. 14.18 *)
  Try   "⟦E,dt⊨c1::√; prg E⊢tn≼C SXcpt Throwable;
          lcl E (EName vn)=None; (prg E,lcl E(EName vn↦Class tn)),dt⊨c2::√⟧ ⟹
                                         E,dt⊨Try c1 Catch(tn vn) c2::√"

  (* cf. 14.18 *)
  Fin   "⟦E,dt⊨c1::√; E,dt⊨c2::√⟧ ⟹
                                         E,dt⊨c1 Finally c2::√"

  Init  "⟦is_class (prg E) C⟧ ⟹
                                         E,dt⊨init C::√"


(* well-typed expressions *)

  (* cf. 15.8 *)
  NewC  "⟦is_class (prg E) C⟧ ⟹
                                         E,dt⊨NewC C::-Class C"
  (* cf. 15.9 *)
  NewA  "⟦is_type (prg E) T;
          E,dt⊨i::-PrimT Integer⟧ ⟹
                                         E,dt⊨New T[i]::-T.[]"
```

```
(* cf. 15.15 *)
Cast  "⟦E,dt⊨e::-T; is_type (prg E) T';
        prg E⊢T≼? T'⟧ ⟹
                                       E,dt⊨Cast T' e::-T'"

(* cf. 15.19.2 *)
Inst  "⟦E,dt⊨e::-RefT T;
        prg E⊢RefT T≼? RefT T'⟧ ⟹
                                       E,dt⊨e InstOf T'::-PrimT Boolean"

(* cf. 15.7.1 *)
Lit   "⟦typeof dt x = Some T⟧ ⟹
                                       E,dt⊨Lit x::-T"

(* cf. 15.10.2, 15.11.1 *)
Super "⟦lcl E This = Some (Class C); C ≠ Object;
        class (prg E) C = Some (D, rest)⟧ ⟹
                                       E,dt⊨Super::-Class D"

(* cf. 15.13.1, 15.10.1, 15.12 *)
Acc   "⟦E,dt⊨va::=T⟧ ⟹
                                       E,dt⊨Acc va::-T"

(* cf. 15.25, 15.25.1 *)
Ass   "⟦E,dt⊨va::=T; va ≠ LVar This;
        E,dt⊨v ::-T';
        prg E⊢T'≼T⟧ ⟹
                                       E,dt⊨va:=v::-T'"

(* cf. 15.24 *)
Cond  "⟦E,dt⊨e0::-PrimT Boolean;
        E,dt⊨e1::-T1; E,dt⊨e2::-T2;
        prg E⊢T1≼T2 ∧ T = T2  ∨  prg E⊢T2≼T1 ∧ T = T1⟧ ⟹
                                       E,dt⊨e0 ? e1 : e2::-T"

(* cf. 15.11.1, 15.11.2, 15.11.3 *)
Call  "⟦E,dt⊨e::-RefT t;
        E,dt⊨ps::≐pTs;
        max_spec (prg E) t (mn, pTs) = {((md,(m,pns,rT)),pTs')}⟧ ⟹
                                E,dt⊨{t,md,invmode m e}e..mn({pTs'}ps)::-rT"

Methd "⟦is_class (prg E) C;
        cmethd (prg E) C sig = Some (md,mh,lvars,blk,res);
        E,dt⊨Body md blk res::-T⟧ ⟹
                                       E,dt⊨Methd C sig::-T"

Body  "⟦is_class (prg E) D;
        E,dt⊨blk::√;
        E,dt⊨res::-T⟧ ⟹
                                       E,dt⊨Body D blk res::-T"
```

```
(* well-typed variables *)

  (* cf. 15.13.1 *)
  LVar  "⟦lcl E vn = Some T; is_type (prg E) T⟧ ⟹
                                          E,dt⊨LVar vn::=T"
  (* cf. 15.10.1 *)
  FVar  "⟦E,dt⊨e::-Class C;
          cfield (prg E) C fn = Some (fd,(m,fT))⟧ ⟹
                                          E,dt⊨{fd,static m}e..fn::=fT"
  (* cf. 15.12 *)
  AVar  "⟦E,dt⊨e::-T.[];
          E,dt⊨i::-PrimT Integer⟧ ⟹
                                          E,dt⊨e.[i]::=T"


(* well-typed expression lists *)

  (* cf. 15.11.??? *)
  Nil                                    "E,dt⊨[]::≐[]"

  (* cf. 15.11.??? *)
  Cons  "⟦E,dt⊨e ::-T;
          E,dt⊨es::≐Ts⟧ ⟹
                                          E,dt⊨e#es::≐T#Ts"

end
```

```
(*  Title:      Isabelle/Bali/WellForm.thy
    ID:         $WellForm.thy,v 1.32 2000/11/27 15:20:16 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Well-formedness of Java programs
for static checks on expressions and statements, see WellType.thy

improvements over Java Specification 1.0 (cf. 8.4.6.3, 8.4.6.4, 9.4.1):
* a method implementing or overwriting another method may have a result type
  that widens to the result type of the other method (instead of identical type)
* if an interface inherits more than one method with the same signature, the
  methods need not have identical return types

simplifications:
* Object and standard exceptions are assumed to be declared like normal classes
*)

WellForm = WellType +

consts

  wf_fdecl :: "prog ⇒          fdecl ⇒ bool"
  wf_mhead :: "prog ⇒ sig   ⇒ mhead ⇒ bool"
  wf_mdecl :: "prog ⇒ tname ⇒ mdecl ⇒ bool"
  wf_idecl :: "prog ⇒          idecl ⇒ bool"
  wf_cdecl :: "prog ⇒          cdecl ⇒ bool"
  wf_prog  :: "prog ⇒                  bool"

defs

  (* well-formed field declaration (common part for classes and interfaces),
     cf. 8.3 and (9.3) *)
  wf_fdecl_def  "wf_fdecl G ≡ λ(fn,(m,ft)). is_type G ft"

  (*well-formed method declaration,cf. 8.4, 8.4.1, 8.4.3, 8.4.5, 14.3.2, (9.4)*)
  (* cf. 14.15, 15.7.2, for scope issues cf. 8.4.1 and 14.3.2 *)
  wf_mhead_def  "wf_mhead G ≡ λ(mn,pTs) (m,pns,rT). length pTs = length pns ∧
                              ( ∀T∈set pTs. is_type G T) ∧ is_type G rT ∧
                              nodups pns"

  wf_mdecl_def  "wf_mdecl G C ≡ λ((mn,pTs),(m,pns,rT),lvars,blk,res).
                 wf_mhead G        (mn,pTs) (m,pns,rT) ∧ unique lvars ∧
                (C=Object ⟶ ¬static m) ∧ (∀(vn,T)∈set lvars. is_type G T) ∧
                (∀pn∈set pns. table_of lvars pn = None) ∧
                (∃T. (G,table_of lvars(pns[↦]pTs) (+)
                        (if static m then empty else empty(()↦Class C)))⊢
                     Body C blk res::-T ∧ G⊢T≼rT)"
```

```
(* well-formed interface declaration, cf. 9.1, 9.1.2.1, 9.1.3, 9.4 *)
wf_idecl_def  "wf_idecl G ≡ λ(I,(si,ms)). ws_idecl G I si ∧
               ¬is_class G I ∧
              (∀(sig,mh)∈set ms. wf_mhead G sig mh ∧ ¬static (fst mh)) ∧
              unique ms ∧
              (o2s ∘ table_of ms hidings Un_tables((λJ.(imethds G J))``set si)
               entails (λmh (md,mh'). G⊢mrt mh≼mrt mh'))"

(* well-formed class declaration, cf. 8.1, 8.1.2.1, 8.1.2.2, 8.1.3, 8.1.4 and
 class method declaration, cf. 8.4.3.3, 8.4.6.1, 8.4.6.2, 8.4.6.3, 8.4.6.4 *)
wf_cdecl_def  "wf_cdecl G ≡ λ(C,(sc,si,fs,ms,init)).
              ¬is_iface G C ∧
              (∀I∈set si. is_iface G I ∧
                   (∀s. ∀(md', mh'   ) ∈ imethds G I s.
                       (∃(md ,(mh ,b)) ∈ cmethd  G C s: G⊢mrt mh≼mrt mh' ∧
                                                ¬static (fst mh)))) ∧
              (∀f∈set fs. wf_fdecl G   f) ∧ unique fs ∧
              (∀m∈set ms. wf_mdecl G C m) ∧ unique ms ∧
              (G,empty)⊢init::√ ∧ ws_cdecl G C sc ∧
              (C ≠ Object ⟶ (table_of ms hiding cmethd G sc entails
                              (λ(mh,b) (md',(mh',b')). G⊢mrt mh≼mrt mh' ∧
                                       static (fst mh') = static (fst mh))))"

(* well-formed program, cf. 8.1, 9.1 *)
wf_prog_def   "wf_prog G ≡ let is = fst G; cs = snd G in
                      ObjectC ∈ set cs  ∧ (∀xn. SXcptC xn ∈ set cs) ∧
              (∀i∈set is. wf_idecl G i) ∧ unique is ∧
              (∀c∈set cs. wf_cdecl G c) ∧ unique cs"

end
```

```
(*  Title:      Isabelle/Bali/State.thy
    ID:         $State.thy,v 1.63 2000/11/23 09:57:31 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

State for evaluation of Java expressions and statements

design issues:
* all kinds of objects (class instances, arrays, and class objects)
  are handeled via a general object abstraction
* the heap and the map for class objects are combined into a single table
  (recall (loc, obj) table × (tname, obj) table  ~=  (loc + tname, obj) table)

simplifications:

*)

State = TypeRel +

datatype  obj_tag =     (* tag for generic object   *)
          CInst tname   (* class instance           *)
        | Arr  ty int   (* array with component type and length *)
     (* | CStat            the tag is irrelevant for a class object,
                           i.e. the static fields of a class *)

types   vn   = "fspec + int"                    (* variable name      *)
        obj  = "obj_tag × (vn, val) table"      (* generalized object *)

constdefs

  the_Arr :: "obj option ⇒ ty × int × (vn, val) table"
 "the_Arr obj ≡ ε(T,k,t). obj = Some (Arr T k,t)"

  upd_obj       :: "vn ⇒ val ⇒ obj ⇒ obj"
 "upd_obj n v ≡ λ(oi,vs). (oi,vs(n↦v))"

  obj_ty        :: "obj ⇒ ty"
 "obj_ty obj    ≡ case fst obj of CInst C ⇒ Class C | Arr T k ⇒ T.[]"

  obj_class :: "obj ⇒ tname"
 "obj_class obj ≡ case fst obj of CInst C ⇒       C | Arr T k ⇒ Object"


types oref = "loc + tname"          (* generalized object reference *)
syntax
  Heap  :: "loc   ⇒ oref"
  Stat  :: "tname ⇒ oref"
translations
  "Heap" => "Inl"
  "Stat" => "Inr"
```

```
constdefs
  fields_table  :: "prog ⇒ tname ⇒ (fspec ⇒ field ⇒ bool)  ⇒ (fspec, ty) table"
 "fields_table G C P ≡ option_map snd ∘ table_of (filter (split P) (fields G C))"

  in_bounds :: "int ⇒ int ⇒ bool"            ("(_/ in'_bounds _)" [50, 51] 50)
 "i in_bounds k ≡ 0 ≤ i ∧ i < k"

  arr_comps :: "'a ⇒ int ⇒ int ⇒ 'a option"
 "arr_comps T k ≡ λi. if i in_bounds k then Some T else None"

  var_tys       :: "prog ⇒ obj_tag ⇒ oref ⇒ (vn, ty) table"
 "var_tys G oi r ≡ case r of Heap a ⇒ (case oi of
           CInst C ⇒ fields_table G C (λn (m,fT). ¬static m) (+) empty
         | Arr T k ⇒ empty (+) arr_comps T k)
                         | Stat C ⇒ fields_table G C
                              (λ(fn,fd) (m,fT). fd = C ∧ static m) (+) empty"


types   globs                  (* global variables: heap and static variables *)
        = "(oref , obj) table"
        heap
        = "(loc  , obj) table"
        locals
        = "(lname, val) table" (* local variables *)


datatype st = (* pure state, i.e. contents of all variables *)
        st globs locals

constdefs

  globs  :: "st ⇒ globs"
 "globs  ≡ st_case (λg l. g)"

  locals :: "st ⇒ locals"
 "locals ≡ st_case (λg l. l)"

  heap   :: "st ⇒ heap"
 "heap s ≡ globs s ∘ Heap"

  new_Addr     :: "heap ⇒ loc option"
 "new_Addr h   ≡ if (∀a. h a ≠ None) then None else Some (εa. h a = None)"

syntax
  val_this     :: "st ⇒ val"
  lookup_obj   :: "st ⇒ val ⇒ obj"
translations
 "val_this s"       == "the (locals s This)"
 "lookup_obj s a'"  == "the (heap s (the_Addr a'))"
```

```
syntax
  init_vals     :: "('a, ty) table ⇒ ('a, val) table"
translations
 "init_vals vs"    == "option_map default_val ∘ vs"

constdefs
  gupd       :: "oref  ⇒ obj ⇒ st ⇒ st"        ("gupd'(_↦_')"[10,10]1000)
 "gupd r obj  ≡ st_case (λg l. st (g(r↦obj)) l)"

  lupd       :: "lname ⇒ val ⇒ st ⇒ st"        ("lupd'(_↦_')"[10,10]1000)
 "lupd vn v   ≡ st_case (λg l. st g (l(vn↦v)))"

  upd_gobj   :: "oref ⇒ vn ⇒ val ⇒ st ⇒ st"
 "upd_gobj r n v ≡ st_case (λg l. st (chg_map (upd_obj n v) r g) l)"

  set_locals  :: "locals ⇒ st ⇒ st"
 "set_locals l ≡ st_case (λg l'. st g l)"

  init_obj    :: "prog ⇒ obj_tag ⇒ oref ⇒ st ⇒ st"
 "init_obj G oi r ≡ gupd(r↦(oi, init_vals (var_tys G oi r)))"

syntax
  init_class_obj :: "prog ⇒ tname ⇒ st ⇒ st"

translations
 "init_class_obj G C" == "init_obj G arbitrary (Inr C)"


datatype xcpt           (* exception *)
        = XcptLoc loc   (* location of allocated execption object *)
        | StdXcpt xname (* intermediate standard exception, see Eval.thy *)

consts

  the_XcptLoc :: "xcpt ⇒ loc"
  the_StdXcpt :: "xcpt ⇒ xname"

defs

  the_XcptLoc_def "the_XcptLoc xc ≡ εa. xc = XcptLoc a"
  the_StdXcpt_def "the_StdXcpt xc ≡ εx. xc = StdXcpt x"

types
  xopt  = "xcpt option"

constdefs
  xcpt_if    :: "bool ⇒ xopt ⇒ xopt ⇒ xopt"
 "xcpt_if c x' x ≡ if c ∧ (x = None) then x' else x"
```

```
syntax
  raise_if :: "bool ⇒ xname ⇒ xopt ⇒ xopt"
  np       :: "val           ⇒ xopt ⇒ xopt"
  check_neg:: "val           ⇒ xopt ⇒ xopt"

translations
 "raise_if c xn" == "xcpt_if c (Some (StdXcpt xn))"
 "np v"          == "raise_if (v = Null)      NullPointer"
 "check_neg i'"  == "raise_if (the_Intg i'<0) NegArrSize"
types
  state = "xopt × st"          (* state including exception information *)

syntax
  Norm     :: "st ⇒ state"

translations
  "Norm s"     == "(None,s)"
  "xopt"       <= (type) "State.xcpt option"
  "xopt"       <= (type) "xcpt option"
  "state"      <= (type) "xopt × State.st"
  "state"      <= (type) "xopt × st"

constdefs
  normal     :: "state ⇒ bool"
 "normal ≡ λs. fst s = None"

  inited   :: "tname ⇒ globs ⇒ bool"
 "inited C g ≡ g (Stat C) ≠ None"

  initd    :: "tname ⇒ state ⇒ bool"
 "initd C ≡ inited C ∘ globs ∘ snd"

  heap_free :: "nat ⇒ state ⇒ bool"
 "heap_free n ≡ λs. atleast_free (heap (snd s)) n"

  xupd     :: "(xopt ⇒ xopt) ⇒ state ⇒ state"
 "xupd f ≡ prod_fun f id"

  supd     :: "(st ⇒ st) ⇒ state ⇒ state"
 "supd ≡ prod_fun id"

syntax
  set_lvars     :: "locals ⇒ state ⇒ state"
  restore_lvars :: "state  ⇒ state ⇒ state"

translations
 "set_lvars l" == "supd (set_locals l)"
 "restore_lvars s' s" == "set_lvars (locals (snd s')) s"

end
```

```
(*  Title:      Isabelle/Bali/Eval.thy
    ID:         $Eval.thy,v 1.85 2000/11/27 15:20:15 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Operational evaluation (big-step) semantics of Java expressions and statements

improvements over Java Specification 1.0:
* dynamic method lookup does not need to consider the return type (cf.15.11.4.4)
* throw raises a NullPointer exception if a null reference is given, and each
  throw of a standard exception yield a fresh exception object (was not specified)
* if there is not enough memory even to allocate an OutOfMemory exception,
  evaluation/execution fails, i.e. simply stops (was not specified)
* array assignment checks lhs (and may throw exceptions) before evaluating rhs
* fixed exact positions of class initializations (immediate at first active use)
design issues:
* evaluation vs. (single-step) transition semantics
  evaluation semantics chosen, because:
  ++ less verbose and therefore easier to read (and to handle in proofs)
  +  more abstract
  +  intermediate values (appearing in recursive rules) need not be stored
     explicitly, e.g. no call body construct or stack of invocation frames
     containing local variables and return addresses for method calls needed
  +  convenient rule induction for subject reduction theorem
  -  no interleaving (for parallelism) can be described
  -  stating a property of infinite executions requires the meta-level argument
     that this property holds for any finite prefixes of it (e.g. stopped using
     a counter that is decremented to zero and then throwing an exception)
* unified evaluation for variables, expressions, expression lists, statements
* the value entry in statement rules is redundant
* the value entry in rules is irrelevant in case of exceptions, but its full
  inclusion helps to make the rule structure independent of exception occurence.
* as irrelevant value entries are ignored, it does not matter if they are unique
  For simplicity, (fixed) arbitrary values are preferred over "free" values.
* the rule format is such that the start state may contain an exception.
  ++ faciliates exception handling
  +  symmetry
* the rules are defined carefully in order to be applicable even in not
  type-correct situations (yielding undefined values),
  e.g. the_Addr (Val (Bool b)) = arbitrary.
  ++ fewer rules
  -  less readable because of auxiliary functions like the_Addr
  Alternative: "defensive" evaluation throwing some InternalError exception
               in case of (impossible, for correct programs) type mismatches
* there is exactly one rule per syntactic construct
  + no redundancy in case distinctions
* halloc fails iff there is no free heap address. When there is
  only one free heap address left, it returns an OutOfMemory exception.
  In this way it is guaranteed that when an OutOfMemory exception is thrown for
  the first time, there is a free location on the heap to allocate it.
```

```
* the allocation of objects that represent standard exceptions is deferred until
  execution of any enclosing catch clause, which is transparent to the program.
  -  requires an auxiliary execution relation
  ++ avoids copies of allocation code and awkward case distinctions (whether
     there is enough memory to allocate the exception) in evaluation rules
* unfortunately new_Addr is not directly executable because of Hilbert operator.

simplifications:
* local variables are initialized with default values (no definite assignment)
* garbage collection not considered, therefore also no finalizers
* stack overflow and memory overflow during class initialization not modelled
* exceptions in initializations not replaced by ExceptionInInitializerError
*)

Eval = State +

types vvar  =         "val × (val ⇒ state ⇒ state)"
      vals  =        "(val, vvar, val list) sum3"
translations
     "vvar" <= (type) "val × (val ⇒ state ⇒ state)"
     "vals" <= (type)"(val, vvar, val list) sum3"

constdefs
  arbitrary3 :: "('al + 'ar, 'b, 'c) sum3 ⇒ vals"
 "arbitrary3 ≡ sum3_case (In1 ∘ sum_case (λx. arbitrary) (λx. Unit))
                     (λx. In2 arbitrary) (λx. In3 arbitrary)"

constdefs
  throw :: "val ⇒ xopt ⇒ xopt"
 "throw a' x ≡ xcpt_if True (Some (XcptLoc (the_Addr a'))) (np a' x)"

  fits    :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool" ("_,_⊢_ fits _"[61,61,61,61]60)
 "G,s⊢a' fits T  ≡ (∃rt. T=RefT rt) ⟶ a'=Null ∨ G⊢obj_ty(lookup_obj s a')≼T"

  catch ::"prog ⇒ state ⇒ tname ⇒ bool"      ("_,_⊢catch _"[61,61,61]60)
 "G,s⊢catch C≡∃xc. fst s=Some xc ∧ G,snd s⊢Addr (the_XcptLoc xc) fits Class C"

  new_xcpt_var :: "ename ⇒ state ⇒ state"
 "new_xcpt_var vn ≡ λ(x,s). Norm(lupd(EName vn↦Addr (the_XcptLoc (the x))) s)"

constdefs

  assign     :: "('a ⇒ state ⇒ state) ⇒ 'a ⇒ state ⇒ state"
 "assign f v ≡ λ(x,s). let (x',s') = (if x = None then f v else id) (x,s)
                   in  (x',if x' = None then s' else s)"

  init_comp_ty :: "ty ⇒ stmt"
 "init_comp_ty T ≡ if (∃C. T = Class C) then init (the_Class T) else Skip"
```

```
constdefs

  target  :: "inv_mode ⇒ st ⇒ val ⇒ ref_ty ⇒ tname"
 "target m s a' t ≡ if m = IntVir
          then obj_class (lookup_obj s a') else the_Class (RefT t)"

  init_lvars :: "prog ⇒ tname ⇒ sig ⇒ inv_mode ⇒ val ⇒ val list ⇒
                 state ⇒ state"
 "init_lvars G C sig mode a' pvs ≡ λ(x,s). let
             (_,(_,pns,_),lvars,_) = the (cmethd G C sig);
             l = init_vals(table_of lvars)(pns[↦]pvs) (+)
                 (if mode=Static then empty else empty(()↦a'))
             in set_lvars l (if mode = Static then x else np a' x,s)"

  body :: "prog ⇒ tname ⇒ sig ⇒ expr"
 "body G C sig ≡ let (D, _, _, c, e) = the (cmethd G C sig) in Body D c e"

consts
  eval   :: "prog ⇒ (state × term    × vals × state) set"
   halloc:: "prog ⇒ (state × obj_tag × loc   × state) set"
  sxalloc:: "prog ⇒ (state                   × state) set"

constdefs
  lvar :: "lname ⇒ st ⇒ vvar"
 "lvar vn s ≡ (the (locals s vn), λv. supd (lupd(vn↦v)))"

  fvar :: "tname ⇒ bool ⇒ ename ⇒ val ⇒ state ⇒ vvar × state"
 "fvar C stat fn a' s ≡ let (oref,xf) = if stat then (Stat C,id)
                                   else (Heap (the_Addr a'),np a');
                     n = Inl (fn,C); f = (λv. supd (upd_gobj oref n v)) in
                 ((the (snd (the (globs (snd s) oref)) n),f),xupd xf s)"

  avar :: "prog ⇒ val ⇒ val ⇒ state ⇒ vvar × state"
 "avar G i' a' s ≡ let oref = Heap (the_Addr a'); i = the_Intg i'; n = Inr i;
                     (T,k,cs) = the_Arr (globs (snd s) oref); f = (λv (x,s).
             (raise_if (¬G,s⊢v fits T) ArrStore x, upd_gobj oref n v s)) in
      ((the (cs n),f), xupd (raise_if (¬i in_bounds k) IndOutBound ∘ np a') s)"

syntax

eval ::"[prog,state,term ,vals*state]=>bool"("_|-_ -_>-> _"  [61,61,80,   61]60)
exec ::"[prog,state,stmt      ,state]=>bool"("_|-_ -_-> _"   [61,61,65,   61]60)
evar ::"[prog,state,var  ,vvar,state]=>bool"("_|-_ -_=>_-> _"[61,61,90,61,61]60)
eval_::"[prog,state,expr ,val, state]=>bool"("_|-_ -_->_-> _"[61,61,80,61,61]60)
evals::"[prog,state,expr list ,
                   val  list ,state]=>bool"("_|-_ -_#>_-> _"[61,61,61,61,61]60)
hallo::"[prog,state,obj_tag,
                    loc,state]=>bool"("_|-_ -halloc _>_-> _"[61,61,61,61,61]60)
sallo::"[prog,state        ,state]=>bool"("_|-_ -sxalloc-> _"[61,61,      61]60)
```

```
syntax (xsymbols)

dummy_res  :: "vals"                          ("•")
eval ::"[prog,state,term,vals×state]⇒bool" ("_⊢_ —_≻→ _" [61,61,80,   61]60)
exec ::"[prog,state,stmt      ,state]⇒bool"("_⊢_ —_→ _"   [61,61,65,   61]60)
evar ::"[prog,state,var  ,vvar,state]⇒bool"("_⊢_ —_=≻_→ _"[61,61,90,61,61]60)
eval_::"[prog,state,expr ,val ,state]⇒bool"("_⊢_ —_-≻_→ _"[61,61,80,61,61]60)
evals::"[prog,state,expr list ,
                    val  list ,state]⇒bool"("_⊢_ —_≐≻_→ _"[61,61,61,61,61]60)
hallo::"[prog,state,obj_tag,
                     loc,state]⇒bool"("_⊢_ —halloc _≻_→ _"[61,61,61,61,61]60)
sallo::"[prog,state,        state]⇒bool"("_⊢_ —sxalloc→ _"[61,61,      61]60)

translations
  "•" == "In1 Unit"
  "G⊢s —t   ≻→  w___s' " == "(s,t,w___s') ∈ eval G"
  "G⊢s —t   ≻→ (w,  s')" <= "(s,t,w,  s') ∈ eval G"
  "G⊢s —t   ≻→ (w,x,s')" <= "(s,t,w,x,s') ∈ eval G"
  "G⊢s —c    →  (x,s')" <= "G⊢s —In1r c≻→ (•     ,x,s')"
  "G⊢s —c    →      s' " == "G⊢s —In1r c≻→ (•     ,  s')"
  "G⊢s —e-≻v →  (x,s')" <= "G⊢s —In1l e≻→ (In1 v ,x,s')"
  "G⊢s —e-≻v →      s' " == "G⊢s —In1l e≻→ (In1 v ,  s')"
  "G⊢s —e=≻vf→  (x,s')" <= "G⊢s —In2  e≻→ (In2 vf,x,s')"
  "G⊢s —e=≻vf→      s' " == "G⊢s —In2  e≻→ (In2 vf,  s')"
  "G⊢s —e≐≻v →  (x,s')" <= "G⊢s —In3  e≻→ (In3 v ,x,s')"
  "G⊢s —e≐≻v →      s' " == "G⊢s —In3  e≻→ (In3 v ,  s')"
  "G⊢s —halloc oi≻a→ (x,s')" <= "(s,oi,a,x,s') ∈ halloc G"
  "G⊢s —halloc oi≻a→    s' " == "(s,oi,a,  s') ∈ halloc G"
  "G⊢s —sxalloc→     (x,s')" <= "(s     ,x,s') ∈ sxalloc G"
  "G⊢s —sxalloc→        s' " == "(s     ,  s') ∈ sxalloc G"

inductive "halloc G" intrs (* allocating objects on the heap, cf. 12.5 *)

  Xcpt  "G⊢(Some x,s) —halloc oi≻arbitrary→ (Some x,s)"

  New   "⟦new_Addr (heap s) = Some a;
         (x,oi') = (if atleast_free (heap s) 2 then (None,oi)
                    else (Some (XcptLoc a),CInst (SXcpt OutOfMemory)))⟧ ⟹
         G⊢Norm s —halloc oi≻a→ (x,init_obj G oi' (Heap a) s)"

inductive "sxalloc G" intrs (* allocating exception objects for
                              standard exceptions (other than OutOfMemory) *)

  Norm   "G⊢ Norm              s   —sxalloc→  Norm             s"

  XcptL  "G⊢(Some (XcptLoc a ),s)  —sxalloc→ (Some (XcptLoc a),s)"

  SXcpt "⟦G⊢Norm s0 —halloc (CInst (SXcpt xn))≻a→ (x,s1)⟧ ⟹
         G⊢(Some (StdXcpt xn),s0) —sxalloc→ (Some (XcptLoc a),s1)"
```

```
inductive "eval G" intrs

(* propagation of exceptions *)

  (* cf. 14.1, 15.5 *)
  Xcpt  "G⊢(Some xc,s) −t≻→ (arbitrary3 t,(Some xc,s))"


(* execution of statements *)

  (* cf. 14.5 *)
  Skip                              "G⊢Norm s −Skip→ Norm s"

  (* cf. 14.7 *)
  Expr  "⟦G⊢Norm s0 −e-≻v→ s1⟧ ⟹
                                  G⊢Norm s0 −Expr e→ s1"

  (* cf. 14.2 *)
  Comp  "⟦G⊢Norm s0 −c1 → s1;
         G⊢     s1 −c2 → s2⟧ ⟹
                                 G⊢Norm s0 −c1;; c2→ s2"


  (* cf. 14.8.2 *)
  If    "⟦G⊢Norm s0 −e-≻b→ s1;
         G⊢     s1−(if the_Bool b then c1 else c2)→ s2⟧ ⟹
                       G⊢Norm s0 −If(e) c1 Else c2 → s2"

  (* cf. 14.10, 14.10.1 *)
  (*      G⊢Norm s0 −If(e) (c;; While(e) c) Else Skip→ s3 *)
  Loop  "⟦G⊢Norm s0 −e-≻b→ s1;
         if the_Bool b then (G⊢s1 −c→ s2 ∧ G⊢s2 −While(e) c→ s3)
                       else s3 = s1⟧ ⟹
                              G⊢Norm s0 −While(e) c→ s3"

  (* cf. 14.16 *)
  Throw "⟦G⊢Norm s0 −e-≻a'→ s1⟧ ⟹
                                 G⊢Norm s0 −Throw e→ xupd (throw a') s1"

  (* cf. 14.18.1 *)
  Try   "⟦G⊢Norm s0 −c1→ s1; G⊢s1 −sxalloc→ s2;
         if G,s2⊢catch C then G⊢new_xcpt_var vn s2 −c2→ s3 else s3 = s2⟧ ⟹
                 G⊢Norm s0 −Try c1 Catch(C vn) c2→ s3"

  (* cf. 14.18.2 *)
  Fin   "⟦G⊢Norm s0 −c1→ (x1,s1);
         G⊢Norm s1 −c2→ s2⟧ ⟹
                  G⊢Norm s0 −c1 Finally c2→ xupd (xcpt_if (x1≠None) x1) s2"
```

```
  (* cf. 12.4.2, 8.5 *)
  Init  "⟦the (class G C) = (sc,si,fs,ms,ini);
          if inited C (globs s0) then s3 = Norm s0
          else (G⊢Norm (init_class_obj G C s0)
                  −(if C = Object then Skip else init sc)→ s1 ∧
                G⊢set_lvars empty s1 −ini→ s2 ∧ s3 = restore_lvars s1 s2)⟧ ⟹
                 G⊢Norm s0 −init C→ s3"


(* evaluation of expressions *)

  (* cf. 15.8.1, 12.4.1 *)
  NewC  "⟦G⊢Norm s0 −init C→ s1;
          G⊢     s1 −halloc (CInst C)≻a→ s2⟧ ⟹
                                 G⊢Norm s0 −NewC C-≻Addr a→ s2"

  (* cf. 15.9.1, 12.4.1 *)
  NewA  "⟦G⊢Norm s0 −init_comp_ty T→ s1; G⊢s1 −e-≻i'→ s2;
          G⊢xupd (check_neg i') s2 −halloc (Arr T (the_Intg i'))≻a→ s3⟧ ⟹
                               G⊢Norm s0 −New T[e]-≻Addr a→ s3"

  (* cf. 15.15 *)
  Cast  "⟦G⊢Norm s0 −e-≻v→ s1;
          s2 = xupd (raise_if (¬G,snd s1⊢v fits T) ClassCast) s1⟧ ⟹
                               G⊢Norm s0 −Cast T e-≻v→ s2"

  (* cf. 15.19.2 *)
  Inst  "⟦G⊢Norm s0 −e-≻v→ s1;
          b = (v≠Null ∧ G,snd s1⊢v fits RefT T)⟧ ⟹
                             G⊢Norm s0 −e InstOf T-≻Bool b→ s1"

  (* cf. 15.7.1 *)
  Lit                             "G⊢Norm s −Lit v-≻v→ Norm s"



  (* cf. 15.10.2 *)
  Super                        "G⊢Norm s −Super-≻val_this s→ Norm s"

  (* cf. 15.2 *)
  Acc   "⟦G⊢Norm s0 −va=≻(v,f)→ s1⟧ ⟹
                                 G⊢Norm s0 −Acc va-≻v→ s1"

  (* cf. 15.25.1 *)
  Ass   "⟦G⊢Norm s0 −va=≻(w,f)→ s1;
          G⊢     s1 −e-≻v  → s2⟧ ⟹
                                  G⊢Norm s0 −va:=e-≻v→ assign f v s2"
```

```
  (* cf. 15.24 *)
  Cond  "⟦G⊢Norm s0 −e0-≻b→ s1;
         G⊢     s1 −(if the_Bool b then e1 else e2)-≻v→ s2⟧ ⟹
                            G⊢Norm s0 −e0 ? e1 : e2-≻v→ s2"

  (* cf. 15.11.4.1, 15.11.4.2, 15.11.4.4, 15.11.4.5 *)
  Call  "⟦G⊢Norm s0 −e-≻a'→ s1; G⊢s1 −args≐≻vs→ s2;
         C = target mode (snd s2) a' cT;
         G⊢init_lvars G C (mn,pTs) mode a' vs s2 −Methd C (mn,pTs)-≻v→ s3⟧ ⟹
         G⊢Norm s0 −{t,cT,mode}e..mn({pTs}args)-≻v→ (restore_lvars s2 s3)"

  Methd "⟦G⊢Norm s0 −body G C sig-≻v→ s1⟧ ⟹
                                G⊢Norm s0 −Methd C sig-≻v→ s1"

  (* cf. 14.15, 12.4.1 *)
  Body  "⟦G⊢Norm s0 −init D→ s1; G⊢s1 −c→ s2; G⊢s2 −e-≻v→ s3⟧ ⟹
                                G⊢Norm s0 −Body D c e-≻v→ s3"


(* evaluation of variables *)

  (* cf. 15.13.1, 15.7.2 *)
  LVar  "G⊢Norm s −LVar vn=≻lvar vn s→ Norm s"

  (* cf. 15.10.1, 12.4.1 *)
  FVar  "⟦G⊢Norm s0 −init C→ s1; G⊢s1 −e-≻a→ s2;
         (v,s2') = fvar C stat fn a s2⟧ ⟹
         G⊢Norm s0 −{C,stat}e..fn=≻v→ s2'"

  (* cf. 15.12.1, 15.25.1 *)
  AVar  "⟦G⊢ Norm s0 −e1-≻a→ s1; G⊢s1 −e2-≻i→ s2;
         (v,s2') = avar G i a s2⟧ ⟹
                      G⊢Norm s0 −e1.[e2]=≻v→ s2'"


(* evaluation of expression lists *)

  (* cf. 15.11.4.2 *)
  Nil
                                   "G⊢Norm s0 −[]≐≻[]→ Norm s0"

  (* cf. 15.6.4 *)
  Cons  "⟦G⊢Norm s0 −e -≻ v → s1;
         G⊢     s1 −es≐≻vs→ s2⟧ ⟹
                                   G⊢Norm s0 −e#es≐≻v#vs→ s2"
monos
  if_def2

end
```

```
(*  Title:      Isabelle/Bali/Evaln.thy
    ID:         $Evaln.thy,v 1.32 2000/11/19 19:09:36 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1999 Technische Universitaet Muenchen

Operational evaluation (big-step) semantics of Java expressions and statements
Variant of eval relation with counter for bounded recursive depth
Evaln could completely replace Eval.
*)
Evaln = Eval +

consts

  evaln :: "prog ⇒ (state × term × nat × vals × state) set"

syntax

  evaln :: "[prog, state, term,        nat, vals * state] => bool"
                               ("_|-_ -_>-_-> _"   [61,61,80,   61,61] 60)
  evarn :: "[prog, state, var  , vvar        , nat, state] => bool"
                               ("_|-_ -_=>-_-> _" [61,61,90,61,61,61] 60)
  eval_n:: "[prog, state, expr , val         , nat, state] => bool"
                               ("_|-_ -_->-_-> _" [61,61,80,61,61,61] 60)
  evalsn:: "[prog, state, expr list, val list, nat, state] => bool"
                               ("_|-_ -_#>-_-> _" [61,61,61,61,61,61] 60)
  execn :: "[prog, state, stmt ,               nat, state] => bool"
                               ("_|-_ -_-_-> _"    [61,61,65,   61,61] 60)

syntax (xsymbols)

  evaln :: "[prog, state, term,         nat, vals × state] ⇒ bool"
                               ("_⊢_ −_≻−_→ _"   [61,61,80,   61,61] 60)
  evarn :: "[prog, state, var  , vvar        , nat, state] ⇒ bool"
                               ("_⊢_ −_=≻_−_→ _" [61,61,90,61,61,61] 60)
  eval_n:: "[prog, state, expr , val ,         nat, state] ⇒ bool"
                               ("_⊢_ −_-≻_−_→ _" [61,61,80,61,61,61] 60)
  evalsn:: "[prog, state, expr list, val  list, nat, state] ⇒ bool"
                               ("_⊢_ −_≐≻_−_→ _" [61,61,61,61,61,61] 60)
  execn :: "[prog, state, stmt ,                nat, state] ⇒ bool"
                               ("_⊢_ −_−_→ _"     [61,61,65,   61,61] 60)
translations

  "G⊢s −t    ≻−n→  w___s' " == "(s,t,n,w___s') ∈ evaln G"
  "G⊢s −t    ≻−n→ (w,  s')" <= "(s,t,n,w,  s') ∈ evaln G"
  "G⊢s −t    ≻−n→ (w,x,s')" <= "(s,t,n,w,x,s') ∈ evaln G"
  "G⊢s −c     −n→ (x,s')" <= "G⊢s −In1r  c≻−n→ (•    ,x,s')"
  "G⊢s −c     −n→    s' " == "G⊢s −In1r  c≻−n→ (•    ,  s')"
  "G⊢s −e-≻v  −n→ (x,s')" <= "G⊢s −In1l e≻−n→ (In1 v ,x,s')"
  "G⊢s −e-≻v  −n→    s' " == "G⊢s −In1l e≻−n→ (In1 v ,  s')"
  "G⊢s −e=≻vf −n→ (x,s')" <= "G⊢s −In2  e≻−n→ (In2 vf,x,s')"
```

```
  "G⊢s −e=≻vf −n→     s' " == "G⊢s −In2  e≻−n→ (In2 vf,  s')"
  "G⊢s −e≐≻v  −n→ (x,s')" <= "G⊢s −In3  e≻−n→ (In3 v ,x,s')"
  "G⊢s −e≐≻v  −n→     s' " == "G⊢s −In3  e≻−n→ (In3 v ,  s')"


inductive "evaln G" intrs

(* propagation of exceptions *)

  Xcpt  "G⊢(Some xc,s) −t≻−n→ (arbitrary3 t,(Some xc,s))"


(* evaluation of variables *)

  LVar  "G⊢Norm s −LVar vn=≻lvar vn s−n→ Norm s"

  FVar  "⟦G⊢Norm s0 −init C−n→ s1; G⊢s1 −e-≻a'−n→ s2;
          (v,s2') = fvar C stat fn a' s2⟧ ⟹
          G⊢Norm s0 −{C,stat}e..fn=≻v−n→ s2'"

  AVar  "⟦G⊢ Norm s0 −e1-≻a−n→ s1 ; G⊢s1 −e2-≻i−n→ s2;
          (v,s2') = avar G i a s2⟧ ⟹
                      G⊢Norm s0 −e1.[e2]=≻v−n→ s2'"




(* evaluation of expressions *)

  NewC  "⟦G⊢Norm s0 −init C−n→ s1;
          G⊢     s1 −halloc (CInst C)≻a→ s2⟧ ⟹
                                    G⊢Norm s0 −NewC C-≻Addr a−n→ s2"

  NewA  "⟦G⊢Norm s0 −init_comp_ty T−n→ s1; G⊢s1 −e-≻i'−n→ s2;
          G⊢xupd (check_neg i') s2 −halloc (Arr T (the_Intg i'))≻a→ s3⟧ ⟹
                                  G⊢Norm s0 −New T[e]-≻Addr a−n→ s3"

  Cast  "⟦G⊢Norm s0 −e-≻v−n→ s1;
          s2 = xupd (raise_if (¬G,snd s1⊢v fits T) ClassCast) s1⟧ ⟹
                                  G⊢Norm s0 −Cast T e-≻v−n→ s2"

  Inst  "⟦G⊢Norm s0 −e-≻v−n→ s1;
          b = (v≠Null ∧ G,snd s1⊢v fits RefT T)⟧ ⟹
                                G⊢Norm s0 −e InstOf T-≻Bool b−n→ s1"

  Lit                         "G⊢Norm s −Lit v-≻v−n→ Norm s"

  Super                       "G⊢Norm s −Super-≻val_this s−n→ Norm s"
```

```
  Acc   "⟦G⊢Norm s0 −va=≻(v,f)−n→ s1⟧ ⟹
                                  G⊢Norm s0 −Acc va-≻v−n→ s1"

  Ass   "⟦G⊢Norm s0 −va=≻(w,f)−n→ s1;
          G⊢      s1 −e-≻v      −n→ s2⟧ ⟹
                                   G⊢Norm s0 −va:=e-≻v−n→ assign f v s2"

  Cond  "⟦G⊢Norm s0 −e0-≻b−n→ s1;
          G⊢      s1 −(if the_Bool b then e1 else e2)-≻v−n→ s2⟧ ⟹
                             G⊢Norm s0 −e0 ? e1 : e2-≻v−n→ s2"

  Call  "⟦G⊢Norm s0 −e-≻a'−n→ s1; G⊢s1 −args≐≻vs−n→ s2;
          C = target mode (snd s2) a' cT;
          G⊢init_lvars G C (mn,pTs) mode a' vs s2 −Methd C (mn,pTs)-≻v−n→ s3⟧
      ⟹ G⊢Norm s0 −{t,cT,mode}e..mn({pTs}args)-≻v−n→ (restore_lvars s2 s3)"

  Methd "⟦G⊢Norm s0 −body G C sig-≻v−n→ s1⟧ ⟹
                                G⊢Norm s0 −Methd C sig-≻v−Suc n→ s1"

  Body  "⟦G⊢Norm s0−init D−n→ s1; G⊢s1 −c−n→ s2; G⊢s2 −e-≻v−n→ s3⟧ ⟹
                                G⊢Norm s0 −Body D c e-≻v−n→ s3"

(* evaluation of expression lists *)

  Nil
                                "G⊢Norm s0 −[]≐≻[]−n→ Norm s0"

  Cons  "⟦G⊢Norm s0 −e -≻ v −n→ s1;
          G⊢      s1 −es≐≻vs−n→ s2⟧ ⟹
                              G⊢Norm s0 −e#es≐≻v#vs−n→ s2"

(* execution of statements *)

  Skip                              "G⊢Norm s −Skip−n→ Norm s"

  Expr  "⟦G⊢Norm s0 −e-≻v−n→ s1⟧ ⟹
                                  G⊢Norm s0 −Expr e−n→ s1"

  Comp  "⟦G⊢Norm s0 −c1 −n→ s1;
          G⊢      s1 −c2 −n→ s2⟧ ⟹
                                 G⊢Norm s0 −c1;; c2−n→ s2"

  If    "⟦G⊢Norm s0 −e-≻b−n→ s1;
          G⊢      s1−(if the_Bool b then c1 else c2)−n→ s2⟧ ⟹
                          G⊢Norm s0 −If(e) c1 Else c2 −n→ s2"

  Loop  "⟦G⊢Norm s0 −e-≻b−n→ s1;
          if the_Bool b then (G⊢s1 −c−n→ s2 ∧ G⊢s2 −While(e) c−n→ s3)
                        else s3 = s1⟧ ⟹
                              G⊢Norm s0 −While(e) c−n→ s3"
```

```
  Throw "⟦G⊢Norm s0 −e-≻a'−n→ s1⟧ ⟹
                              G⊢Norm s0 −Throw e−n→ xupd (throw a') s1"

  Try   "⟦G⊢Norm s0 −c1−n→ s1; G⊢s1 −sxalloc→ s2;
         if G,s2⊢catch tn then G⊢new_xcpt_var vn s2 −c2−n→ s3 else s3 = s2⟧ ⟹
                G⊢Norm s0 −Try c1 Catch(tn vn) c2−n→ s3"

  Fin   "⟦G⊢Norm s0 −c1−n→ (x1,s1);
         G⊢Norm s1 −c2−n→ s2⟧ ⟹
                 G⊢Norm s0 −c1 Finally c2−n→ xupd (xcpt_if (x1≠None) x1) s2"

  Init  "⟦the (class G C) = (sc,si,fs,ms,ini);
         if inited C (globs s0) then s3 = Norm s0
         else (G⊢Norm (init_class_obj G C s0)
                −(if C = Object then Skip else init sc)−n→ s1 ∧
               G⊢set_lvars empty s1 −ini−n→ s2 ∧ s3 = restore_lvars s1 s2)⟧ ⟹
               G⊢Norm s0 −init C−n→ s3"
monos
  if_def2

end
```

```
(*  Title:      Isabelle/Bali/Conform.thy
    ID:         $Conform.thy,v 1.5 2000/10/19 21:21:51 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

Conformance notions for the type soundness proof for Java

design issues:
* lconf allows for (arbitrary) inaccessible values
* "conforms" does not directly imply that the dynamic types of all objects on
  the heap are indeed existing classes. Yet this can be inferred for all
  referenced objs.
*)

Conform = State +

types   env_ = "prog × (lname, ty) table" (* same as env of WellType.thy *)

constdefs

  gext    :: "st ⇒ st ⇒ bool"              ("_≤|_"         [71,71]   70)
   "s≤|s' ≡ ∀r. ∀(oi, fs)∈globs s r: ∃(oi',fs')∈globs s' r: oi' = oi"

  conf  :: "prog ⇒ st ⇒ val ⇒ ty ⇒ bool"  ("_,_⊢_::≼_"    [71,71,71,71] 70)
          "G,s⊢v::≼T ≡ ∃T'∈typeof (λa. option_map obj_ty (heap s a)) v:G⊢T'≼T"

  lconf :: "prog ⇒ st ⇒ ('a, val) table ⇒ ('a, ty) table ⇒ bool"
                                          ("_,_⊢_[::≼]_" [71,71,71,71] 70)
          "G,s⊢vs[::≼]Ts ≡ ∀n. ∀T∈Ts n: ∃v∈vs n: G,s⊢v::≼T"

  oconf :: "prog ⇒ st ⇒ obj ⇒ oref ⇒ bool"  ("_,_⊢_::≼√_" [71,71,71,71] 70)
          "G,s⊢obj::≼√r ≡ G,s⊢snd obj[::≼]var_tys G (fst obj) r ∧ (case r of
                             Heap a ⇒ is_type G (obj_ty obj) | Stat C ⇒ True)"

  conforms :: "state ⇒ env_ ⇒ bool"          (     "_::≼_"  [71,71]       70)
              "xs::≼E ≡ let (G, L) = E; s = snd xs; l = locals s in
          (∀r. ∀obj∈globs s r:           G,s⊢obj   ::≼√r) ∧
                                          G,s⊢l    [::≼]L   ∧
          (∀a. fst xs=Some(XcptLoc a) ⟶ G,s⊢Addr a::≼Class (SXcpt Throwable))"

end
```

```
(*  Title:      Isabelle/Bali/TypeSafe.thy
    ID:         $TypeSafe.thy,v 1.21 2000/11/25 23:58:27 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

The type soundness proof for Java
*)

TypeSafe = Eval + WellForm + Conform +

constdefs
  DynT_prop::"[prog,inv_mode,tname,ref_ty] ⇒ bool" ("_⊢_→_≼_"[71,71,71,71] 70)
 "G⊢mode→D≼t ≡ mode = IntVir ⟶ is_class G D ∧
                     (if (∃T. t=ArrayT T) then D=Object else G⊢Class D≼RefT t)"

  assign_conforms :: "st ⇒ (val ⇒ state ⇒ state) ⇒ ty ⇒ env ⇒ bool"
         ("_≤|_≼_::≼_"                                        [71,71,71,71] 70)
 "s≤|f≼T::≼E ≡
  ∀s' w. Norm s'::≼E ⟶ fst E,s'⊢w::≼T ⟶ s≤|s' ⟶ assign f w (Norm s')::≼E"

  rconf :: "prog ⇒ lenv ⇒ st ⇒ term ⇒ vals ⇒ tys ⇒ bool"
         ("_,_,_⊢_≻_::≼_"                                [71,71,71,71,71,71] 70)
          "G,L,s⊢t≻v::≼T ≡ case T of
    Inl T  ⇒ if (∃vf. t=In2 vf)
             then G,s⊢fst (the_In2 v)::≼T ∧ s≤|snd (the_In2 v)≼T::≼(G,L)
             else G,s⊢the_In1 v::≼T
  | Inr Ts ⇒ list_all2 (conf G s) (the_In3 v) Ts"

end
```

```
(*  Title:      Isabelle/Bali/AxSem.thy
    ID:         $AxSem.thy,v 1.109 2000/11/25 23:58:27 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1998 Technische Universitaet Muenchen

Axiomatic semantics of Java expressions and statements (see also Eval.thy)

design issues:
* a strong version of validity for triples with premises, namely one that takes
  the recursive depth needed to complete execution, enables correctness proof
* auxiliary variables are handled first-class (-> Thomas Kleymann)
* expressions not flattened to elementary assignments (as usual for axiomatic
  semantics) but treated first-class => explicit result value handling
* intermediate values not on triple, but on assertion level (with result entry)
* multiple results with semantical substitution mechnism not requiring a stack
* because of dynamic method binding, terms need to be dependent on state.
  this is also useful for conditional expressions and statements
* result values in triples exactly as in eval relation (also for xcpt states)
* validity: additional assumption of state conformance and well-typedness,
  which is required for soundness and thus rule hazard required of completeness

restrictions:
* all triples in a derivation are of the same type (due to weak polymorphism)
*)

AxSem = Evaln + TypeSafe +

types  res = vals (* result entry *)

syntax
  Val  :: "val      ⇒ res"
  Var  :: "var      ⇒ res"
  Vals :: "val list ⇒ res"
translations
  "Val  x"     => "(In1 x)"
  "Var  x"     => "(In2 x)"
  "Vals x"     => "(In3 x)"

syntax
  "Val_"    :: [pttrn] => pttrn     ("Val:_"  [951] 950)
  "Var_"    :: [pttrn] => pttrn     ("Var:_"  [951] 950)
  "Vals_"   :: [pttrn] => pttrn     ("Vals:_" [951] 950)
translations
  "λVal:v . b"  == "(λv. b) ∘ the_In1"
  "λVar:v . b"  == "(λv. b) ∘ the_In2"
  "λVals:v. b"  == "(λv. b) ∘ the_In3"
```

```
(* relation on result value, state and auxiliary variables *)
types 'a assn   =        "res ⇒ state ⇒ 'a ⇒ bool"

translations
      "res"    <= (type) "AxSem.res"
      "a assn" <= (type) "vals ⇒ state ⇒ a ⇒ bool"

constdefs
  assn_imp   :: "'a assn ⇒ 'a assn ⇒ bool"             (infixr "⇒" 25)
 "P ⇒ Q ≡ ∀Y s Z. P Y s Z ⟶ Q Y s Z"

  peek_and   :: "'a assn ⇒ (state ⇒  bool) ⇒ 'a assn" (infixl "∧." 13)
 "P ∧. p ≡ λY s Z. P Y s Z ∧ p s"

  assn_supd  :: "'a assn ⇒ (state ⇒ state) ⇒ 'a assn" (infixl ";." 13)
 "P ;. f ≡ λY s' Z. ∃s. P Y s Z ∧ s' = f s"

  supd_assn  :: "(state ⇒ state) ⇒ 'a assn ⇒ 'a assn" (infixr ".;" 13)
 "f .; P ≡ λY s. P Y (f s)"

  subst_res   :: "'a assn ⇒ res ⇒ 'a assn"              ("_←_"  [60,61] 60)
 "P←w ≡ λY. P w"

  subst_Bool  :: "'a assn ⇒ bool ⇒ 'a assn"             ("_←=_" [60,61] 60)
 "P←=b ≡ λY s Z. ∃v. P (Val v) s Z ∧ (normal s ⟶ the_Bool v=b)"

  peek_res    :: "(res ⇒ 'a assn) ⇒ 'a assn"
 "peek_res Pf ≡ λY. Pf Y Y"

  peek_st    :: "(st ⇒ 'a assn) ⇒ 'a assn"
 "peek_st P ≡ λY s. P (snd s) Y s"

  ign_res    ::  "        'a assn ⇒ 'a assn"             ("_↓" [1000] 1000)
  "P↓        ≡ λY s Z. ∃Y. P Y s Z"

syntax
  Normal     ::          "'a assn ⇒ 'a assn"
"@peek_res"  :: "pttrn ⇒ 'a assn ⇒ 'a assn"             ("λ_:. _" [0,3] 3)
"@peek_st"   :: "pttrn ⇒ 'a assn ⇒ 'a assn"             ("λ_.. _" [0,3] 3)

translations
  "Normal P" == "P ∧. normal"
  "λw:. P"   == "peek_res (λw. P)"
  "λs.. P"   == "peek_st (λs. P)"

constdefs
  ign_res_eq :: "'a assn ⇒ res ⇒ 'a assn"              ("_↓=_"  [60,61] 60)
 "P↓=w       ≡ λY:. P↓ ∧. (λs. Y=w)"
```

```
  RefVar     :: "(state ⇒ vvar × state) ⇒ 'a assn ⇒ 'a assn" (infixr "..;" 13)
 "vf ..; P ≡ λY s. let (v,s') = vf s in P (Var v) s'"

  Alloc      :: "prog ⇒ obj_tag ⇒ 'a assn ⇒ 'a assn"
 "Alloc G otag P ≡ λY s Z.
                  ∀s' a. G⊢s −halloc otag≻a→ s'⟶ P (Val (Addr a)) s' Z"

  SXAlloc    :: "prog ⇒ 'a assn ⇒ 'a assn"
 "SXAlloc G P ≡ λY s Z. ∀s'. G⊢s −sxalloc→ s' ⟶ P Y s' Z"

  type_ok  :: "prog ⇒ term ⇒ state ⇒ bool"
 "type_ok G t s ≡ ∃L T. (normal s ⟶ (G,L)⊢t::T) ∧ s::≼(G,L)"


datatype    'a triple =      triple ('a assn) term ('a assn) (** should be
something like triple = ∀'a. triple ('a assn) term ('a assn)  **)

                                          ("{(1_)}/ _>/ {(1_)}"      [3,65,3] 75)
types    'a triples = 'a triple set

syntax

  var_triple   :: "['a assn, var         ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _=>/ {(1_)}"    [3,80,3] 75)
  expr_triple  :: "['a assn, expr        ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _->/ {(1_)}"    [3,80,3] 75)
  exprs_triple :: "['a assn, expr list   ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _#>/ {(1_)}"    [3,65,3] 75)
  stmt_triple  :: "['a assn, stmt,        'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ ._./ {(1_)}"    [3,65,3] 75)

syntax (xsymbols)

  triple       :: "['a assn, term        ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _≻/ {(1_)}"     [3,65,3] 75)
  var_triple   :: "['a assn, var         ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _=≻/ {(1_)}"    [3,80,3] 75)
  expr_triple  :: "['a assn, expr        ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _-≻/ {(1_)}"    [3,80,3] 75)
  exprs_triple :: "['a assn, expr list   ,'a assn] ⇒ 'a triple"
                                          ("{(1_)}/ _≐≻/ {(1_)}"    [3,65,3] 75)

translations
  "{P} e-≻ {Q}" == "{P} In1l e≻ {Q}"
  "{P} e=≻ {Q}" == "{P} In2  e≻ {Q}"
  "{P} e≐≻ {Q}" == "{P} In3  e≻ {Q}"
  "{P} .c. {Q}" == "{P} In1r c≻ {Q}"
```

```
constdefs
  mtriples  :: "('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c ⇒ 'sig ⇒ expr) ⇒
               ('c ⇒ 'sig ⇒ 'a assn) ⇒ ('c ×  'sig) set ⇒ 'a triples"
                                   ("{{(1_)}/ _-≻/ {(1_)} | _}"[3,65,3,65]75)
 "{{P} tf-≻ {Q} | ms} ≡ (λ(C,sig). {Normal(P C sig)} tf C sig-≻ {Q C sig})``ms"

consts

 triple_valid :: "prog ⇒ nat ⇒         'a triple  ⇒ bool"
                                                (   "_⊨_:_"  [61,0, 58] 57)
    ax_valids :: "prog ⇒ 'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_|⊨_"    [61,58,58] 57)
    ax_derivs :: "prog ⇒ ('b triples × 'a triples) set"

syntax

 triples_valid:: "prog ⇒ nat ⇒         'a triples ⇒ bool"
                                                (  "_||=_:_" [61,0, 58] 57)
     ax_valid :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ( "_,_|=_"   [61,58,58] 57)
     ax_Derivs:: "prog ⇒  'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_||-_"   [61,58,58] 57)
     ax_Deriv :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ( "_,_|-_"   [61,58,58] 57)

syntax (xsymbols)

 triples_valid:: "prog ⇒ nat ⇒         'a triples ⇒ bool"
                                                (  "_|⊨_:_"  [61,0, 58] 57)
     ax_valid :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ( "_,_⊨_"    [61,58,58] 57)
     ax_Derivs:: "prog ⇒  'b triples ⇒ 'a triples ⇒ bool"
                                                ("_,_|⊢_"    [61,58,58] 57)
     ax_Deriv :: "prog ⇒  'b triples ⇒ 'a triple  ⇒ bool"
                                                ( "_,_⊢_"    [61,58,58] 57)

defs  triple_valid_def   "G⊨n:t  ≡ case t of {P} t≻ {Q} ⇒
                          ∀Y s Z. P Y s Z ⟶ type_ok G t s ⟶
                          (∀Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z)"

translations         "G|⊨n:ts" == "Ball ts (triple_valid G n)"
defs   ax_valids_def "G,A|⊨ts  ≡  ∀n. G|⊨n:A ⟶ G|⊨n:ts"
translations         "G,A ⊨t"  == "G,A|⊨{t}"
                     "G,A|⊢ts" == "(A,ts) ∈ ax_derivs G"
                     "G,A ⊢t"  == "G,A|⊢{t}"
```

```
inductive "ax_derivs G" intrs

  empty " G,A|⊢{}"
  insert"⟦G,A⊢t; G,A|⊢ts⟧ ⟹
          G,A|⊢insert t ts"

  asm    "ts⊆A ⟹ G,A|⊢ts"

(* could be added for convenience and efficiency, but is not necessary
  cut    "⟦G,A'|⊢ts; G,A|⊢A'⟧ ⟹
           G,A |⊢ts"
*)
  weaken"⟦G,A|⊢ts'; ts ⊆ ts'⟧ ⟹ G,A|⊢ts"

  conseq"∀Y s Z . P  Y s Z  ⟶ (∃P' Q'. G,A⊢{P'} t≻ {Q'} ∧ (∀Y' s'.
        (∀Y   Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶
                              Q  Y' s' Z ))
                                        ⟹ G,A⊢{P } t≻ {Q }"

  hazard"G,A⊢{P ∧. Not ∘ type_ok G t} t≻ {Q}"

  Xcpt  "G,A⊢{P←(arbitrary3 t) ∧. Not ∘ normal} t≻ {P}"

  (* variables *)
  LVar  " G,A⊢{Normal (λs.. P←Var (lvar vn s))} LVar vn=≻ {P}"

  FVar  "⟦G,A⊢{Normal P} .init C. {Q};
          G,A⊢{Q} e-≻ {λVal:a:. fvar C stat fn a ..; R}⟧ ⟹
                              G,A⊢{Normal P} {C,stat}e..fn=≻ {R}"

  AVar  "⟦G,A⊢{Normal P} e1-≻ {Q};
          ∀a. G,A⊢{Q←Val a} e2-≻ {λVal:i:. avar G i a ..; R}⟧ ⟹
                              G,A⊢{Normal P} e1.[e2]=≻ {R}"
  (* expressions *)

  NewC  "⟦G,A⊢{Normal P} .init C. {Alloc G (CInst C) Q}⟧ ⟹
                              G,A⊢{Normal P} NewC C-≻ {Q}"

  NewA  "⟦G,A⊢{Normal P} .init_comp_ty T. {Q};  G,A⊢{Q} e-≻
          {λVal:i:. xupd (check_neg i) .; Alloc G (Arr T (the_Intg i)) R}⟧ ⟹
                              G,A⊢{Normal P} New T[e]-≻ {R}"

  Cast  "⟦G,A⊢{Normal P} e-≻ {λVal:v:. λs..
          xupd (raise_if (¬G,s⊢v fits T) ClassCast) .; Q←Val v}⟧ ⟹
                              G,A⊢{Normal P} Cast T e-≻ {Q}"

  Inst  "⟦G,A⊢{Normal P} e-≻ {λVal:v:. λs..
                  Q←Val (Bool (v≠Null ∧ G,s⊢v fits RefT T))}⟧ ⟹
                              G,A⊢{Normal P} e InstOf T-≻ {Q}"
```

```
Lit                            "G,A⊢{Normal (P←Val v)} Lit v-≻ {P}"

Super " G,A⊢{Normal (λs.. P←Val (val_this s))} Super-≻ {P}"

Acc   "⟦G,A⊢{Normal P} va=≻ {λVar:(v,f):. Q←Val v}⟧ ⟹
                                G,A⊢{Normal P} Acc va-≻ {Q}"

Ass   "⟦G,A⊢{Normal P} va=≻ {Q};
   ∀vf. G,A⊢{Q←Var vf} e-≻ {λVal:v:. assign (snd vf) v .; R}⟧ ⟹
                                G,A⊢{Normal P} va:=e-≻ {R}"

Cond  "⟦G,A ⊢{Normal P} e0-≻ {P'};
        ∀b. G,A⊢{P'←=b} (if b then e1 else e2)-≻ {Q}⟧ ⟹
                                G,A⊢{Normal P} e0 ? e1 : e2-≻ {Q}"

Call  "⟦G,A⊢{Normal P} e-≻ {Q}; ∀a. G,A⊢{Q←Val a} args≐≻ {R a};
        ∀a vs D l. G,A⊢{(R a←Vals vs ∧.
             (λs. D = target mode (snd s) a cT ∧ l = locals (snd s)) ;.
             init_lvars G D (mn,pTs) mode a vs) ∧.
             (λs. normal s ⟶ G⊢mode→D≼t)}
            Methd D (mn,pTs)-≻ {set_lvars l .; S}⟧ ⟹
       G,A⊢{Normal P} {t,cT,mode}e..mn({pTs}args)-≻ {S}"

Methd "⟦G,A∪ {{P} Methd-≻ {Q} | ms} |⊢ {{P} body G-≻ {Q} | ms}⟧ ⟹
                                G,A|⊢{{P} Methd-≻  {Q} | ms}"

Body  "⟦G,A⊢{Normal P} .init D. {Q}; G,A⊢{Q} .c. {R}; G,A⊢{R} e-≻ {S}⟧ ⟹
                                G,A⊢{Normal P} Body D c e-≻ {S}"


(* expression lists *)

Nil                            "G,A⊢{Normal (P←Vals [])} []≐≻ {P}"

Cons  "⟦G,A⊢{Normal P} e-≻ {Q};
        ∀v. G,A⊢{Q←Val v} es≐≻ {λVals:vs:. R←Vals (v#vs)}⟧ ⟹
                                G,A⊢{Normal P} e#es≐≻ {R}"


(* statements *)

Skip                           "G,A⊢{Normal (P←•)} .Skip. {P}"

Expr  "⟦G,A⊢{Normal P} e-≻ {Q←•}⟧ ⟹
                                G,A⊢{Normal P} .Expr e. {Q}"

Comp  "⟦G,A⊢{Normal P} .c1. {Q};
        G,A⊢{Q} .c2. {R}⟧ ⟹
                                G,A⊢{Normal P} .c1;;c2. {R}"
```

```
  If    "⟦G,A ⊢{Normal P} e-≻ {P'};
          ∀b. G,A⊢{P'←=b} .(if b then c1 else c2). {Q}⟧ ⟹
                                 G,A⊢{Normal P} .If(e) c1 Else c2. {Q}"
(* unfolding variant of Loop, not needed here
  LoopU "⟦G,A ⊢{Normal P} e-≻ {P'};
          ∀b. G,A⊢{P'←=b} .(if b then c;;While(e) c else Skip).{Q}⟧
         ⟹             G,A⊢{Normal P} .While(e) c. {Q}"
*)
  Loop  "⟦G,A⊢{P} e-≻ {P'}; G,A⊢{Normal (P'←=True)} .c. {P}⟧ ⟹
                            G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"

  Throw "⟦G,A⊢{Normal P} e-≻ {λVal:a:. xupd (throw a) .; Q←•}⟧ ⟹
                                 G,A⊢{Normal P} .Throw e. {Q}"

  Try   "⟦G,A⊢{Normal P} .c1. {SXAlloc G Q};
          G,A⊢{Q ∧. (λs.  G,s⊢catch C) ;. new_xcpt_var vn} .c2. {R};
              (Q ∧. (λs. ¬G,s⊢catch C)) ⇒ R⟧ ⟹
                                 G,A⊢{Normal P} .Try c1 Catch(C vn) c2. {R}"

  Fin   "⟦G,A⊢{Normal P} .c1. {Q};
      ∀x. G,A⊢{Q ∧. (λs. x = fst s) ;. xupd (λx. None)}
              .c2. {xupd (xcpt_if (x≠None) x) .; R}⟧ ⟹
                                 G,A⊢{Normal P} .c1 Finally c2. {R}"

  Done                         "G,A⊢{Normal (P←• ∧. initd C)} .init C. {P}"

  Init  "⟦the (class G C) = (sc,si,fs,ms,ini);
          G,A⊢{Normal ((P ∧. Not ∘ initd C) ;. supd (init_class_obj G C))}
              .(if C = Object then Skip else init sc). {Q};
      ∀l. G,A⊢{Q ∧. (λs. l = locals (snd s)) ;. set_lvars empty}
              .ini. {set_lvars l .; R}⟧ ⟹
                               G,A⊢{Normal (P ∧. Not ∘ initd C)} .init C. {R}"

rules (** these terms are the same as above, but with generalized typing **)

  polymorphic_conseq
        "∀Y s Z . P  Y s Z  ⟶ (∃P' Q'. G,A⊢{P'} t≻ {Q'} ∧ (∀Y' s'.
        (∀Y   Z'. P' Y s Z' ⟶ Q' Y' s' Z') ⟶
                                Q  Y' s' Z ))
                                         ⟹ G,A⊢{P } t≻ {Q }"

  polymorphic_Loop
        "⟦G,A⊢{P} e-≻ {P'}; G,A⊢{Normal (P'←=True)} .c. {P}⟧ ⟹
                            G,A⊢{P} .While(e) c. {(P'←=False)↓=•}"

end
```

```
(*  Title:      Isabelle/Bali/AxSound.thy
    ID:         $AxSound.thy,v 1.11 2000/11/19 19:09:35 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1999 Technische Universitaet Muenchen

Soundness proof for Axiomatic semantics of Java expressions and statements
*)

AxSound = AxSem +

consts

 triple_valid2:: "prog ⇒ nat ⇒        'a triple  ⇒ bool"
                                                (   "_|=_::_"[61,0, 58] 57)
    ax_valids2:: "prog ⇒ 'a triples ⇒ 'a triples ⇒ bool"
                                                ("_,_||=::_" [61,58,58] 57)

defs  triple_valid2_def  "G|=n::t ≡ case t of {P} t≻ {Q} ⇒
 ∀Y s Z. P Y s Z ⟶ (∀L. s::≼(G,L) ⟶ (∀T. (normal s ⟶ (G,L)⊢t::T) ⟶
 (∀Y' s'. G⊢s −t≻−n→ (Y',s') ⟶ Q Y' s' Z ∧ s'::≼(G,L))))"
defs  ax_valids2_def  "G,A||=::ts ≡  ∀n. (∀t∈A. G|=n::t) ⟶ (∀t∈ts. G|=n::t)"

end
```

```
(*  Title:      Isabelle/Bali/AxCompl.thy
    ID:         $AxCompl.thy,v 1.32 2000/11/19 19:09:34 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1999 Technische Universitaet Muenchen

Completeness proof for Axiomatic semantics of Java expressions and statements

design issues:
* proof structured by Most General Formulas (-> Thomas Kleymann)
*)

AxCompl = AxSem +

constdefs

  nyinitcls :: "prog ⇒ state ⇒ tname set"
 "nyinitcls G s ≡ {C. is_class G C ∧ ¬ initd C s}"

  init_le :: "prog ⇒ nat ⇒ state ⇒ bool"           ("_⊢init≤_"  [51,51] 50)
 "G⊢init≤n ≡ λs. card (nyinitcls G s) ≤ n"

consts (* Most General Triples and Formulas *)

  remember_init_state :: "state assn"               ("≐")
  MGF ::"[state assn, term, prog] ⇒ state triple"   ("{_} _≻ {_→}"[3,65,3]62)
  MGFn::"[nat       , term, prog] ⇒ state triple" ("{=:_} _≻ {_→}"[3,65,3]62)


defs

  remember_init_state_def "≐ ≡ λY s Z. s = Z"

  MGF_def
  "{P} t≻ {G→} ≡ {P} t≻ {λY s' s. G⊢s −t≻→ (Y,s')}"

  MGFn_def
  "{=:n} t≻ {G→} ≡ {≐ ∧. G⊢init≤n} t≻ {G→}"

end
```

```
(*  Title:      Isabelle/Bali/Example.thy
    ID:         $Example.thy,v 1.38 2000/11/23 09:57:30 oheimb Exp $
    Author:     David von Oheimb
    Copyright   1997 Technische Universitaet Muenchen

The following example Bali program includes:
* class and interface declarations with inheritance, hiding of fields,
    overriding of methods (with refined result type), array type,
* method call (with dynamic binding), parameter access, return expressions,
* expression statements, sequential composition, literal values,
    local assignment, local access, field assignment, type cast,
* exception generation and propagation, try & catch statement, throw statement
* instance creation and (default) static initialization


interface HasFoo {
  public Base foo(Base z);
}

class Base implements HasFoo {
  static boolean arr[] = new boolean[2];
  HasFoo vee;
  public Base foo(Base z) {
    return z;
  }
}

class Ext extends Base {
  int vee;
  public Ext foo(Base z) {
    ((Ext)z).vee = 1;
    return null;
  }
}

class Example {
  public static void main(String args[]) throws Throwable {
    Base e = new Ext();
    try {e.foo(null); }
    catch(NullPointerException z) {
      while(Ext.arr[2]) ;
    }
  }
}
*)

Example = Eval + WellForm +

datatype tnam_ = HasFoo_ | Base_ | Ext_ (** cannot simply instantiate tnam **)
datatype enam_ = arr_ | vee_ | z_ | e_
```

```
consts

  tnam_ :: "tnam_ ⇒ tnam"
  enam_ :: "enam_ ⇒ ename"

rules  (** tnam_ and enam_ are intended to be isomorphic to tnam and ename **)

  inj_tnam_  "(tnam_ x = tnam_ y) = (x = y)"
  inj_enam_  "(enam_ x = enam_ y) = (x = y)"

  surj_tnam_ "∃m. n = tnam_ m"
  surj_enam_ "∃m. n = enam_ m"

defs

  Object_mdecls_def "Object_mdecls ≡ []"
  SXcpt_mdecls_def  "SXcpt_mdecls  ≡ []"

syntax

  HasFoo, Base, Ext :: tname
  arr, vee, z, e    :: ename

translations

  "HasFoo" == "TName (tnam_ HasFoo_)"
  "Base"   == "TName (tnam_ Base_)"
  "Ext"    == "TName (tnam_ Ext_)"
  "arr"    ==        "enam_ arr_"
  "vee"    ==        "enam_ vee_"
  "z"      ==        "enam_ z_"
  "e"      ==        "enam_ e_"

consts

  foo    :: mname

constdefs

  foo_sig   :: sig
 "foo_sig   ≡ (foo,[Class Base])"

  foo_mhead :: mhead
 "foo_mhead ≡ (False,[z],Class Base)"

constdefs

  Base_foo :: mdecl
 "Base_foo ≡ (foo_sig, (foo_mhead,([],Skip,!!z)))"
```

```
  Ext_foo  :: mdecl
 "Ext_foo  ≡ (foo_sig, ((False,[z],Class Ext),
             ([],Expr({Ext,False}Cast (Class Ext) (!!z)..vee :=
                      Lit (Intg #1)),Lit Null)
             ))"

  arr_viewed_from :: "tname ⇒ var"
 "arr_viewed_from C ≡ {Base,True}StatRef (ClassT C)..arr"

constdefs

  HasFooInt :: iface
 "HasFooInt ≡ ([], [(foo_sig, foo_mhead)])"

  BaseCl :: class
 "BaseCl ≡ (Object, [HasFoo],
           [(arr, (True,  PrimT Boolean.[])),
            (vee, (False, Iface HasFoo    ))],
           [Base_foo],
           Expr(arr_viewed_from Base := New (PrimT Boolean)[Lit (Intg #2)]))"

  ExtCl  :: class
 "ExtCl  ≡ (Base  , [],
           [(vee, (False, PrimT Integer  ))],
           [Ext_foo],
           Skip)"

constdefs

  ifaces :: idecl list
 "ifaces ≡ [(HasFoo,HasFooInt)]"

  clsses :: cdecl list (** name not 'classes' because of clash with thy token **)
 "clsses ≡ [(Base,BaseCl),(Ext,ExtCl)]@standard_classes"

  test    :: "(ty)list ⇒ stmt"
 "test pTs ≡ e:==NewC Ext;;
             Try Expr({ClassT Base,ClassT Base,IntVir}!!e..
                     foo({pTs}[Lit Null]))
             Catch((SXcpt NullPointer) z)
              (While(Acc (Acc (arr_viewed_from Ext).[Lit (Intg #2)])) Skip)"

consts
  a,b,c   :: loc
syntax
 "classes"            :: cdecl list
  tprg                :: prog

  obj_a, obj_b, obj_c :: obj
  arr_N, arr_a        :: (vn, val) table
```

```
  globs1,globs2,
  globs3,globs8       :: globs
  locs3,locs4,locs8   :: locals
  s0,s0',s9',
  s1,s1',s2,s2',
  s3,s3',s4,s4',
  s6',s7',s8,s8'      :: state

translations

  "classes" == "clsses"
  "tprg"    == "(ifaces,classes)"

  "obj_a"   <= "(Arr (PrimT Boolean) #2, empty(Inr #0↦Bool False)(Inr #1↦Bool False))"
  "obj_b"   <= "(CInst Ext,(empty(Inl (vee, Base)↦Null   )
                                 (Inl (vee, Ext )↦Intg #0)))"
  "obj_c"   == "(CInst (SXcpt NullPointer),empty)"
  "arr_N"   == "empty(Inl (arr, Base)↦Null)"
  "arr_a"   == "empty(Inl (arr, Base)↦Addr a)"
  "globs1"  == "empty(Inr Ext   ↦(arbitrary, empty))
                     (Inr Base  ↦(arbitrary, arr_N))
                     (Inr Object↦(arbitrary, empty))"
  "globs2"  == "empty(Inr Ext   ↦(arbitrary, empty))
                     (Inr Object↦(arbitrary, empty))
                     (Inl a↦obj_a)
                     (Inr Base  ↦(arbitrary, arr_a))"
  "globs3"  == "globs2(Inl b↦obj_b)"
  "globs8"  == "globs3(Inl c↦obj_c)"
  "locs3"   == "empty(Inl e↦Addr b)"
  "locs4"   == "empty(Inl z↦Null)(Inr()↦Addr b)"
  "locs8"   == "locs3(Inl z↦Addr c)"
  "s0"  == "       st empty  empty"
  "s0'" == " Norm  s0"
  "s1"  == "       st globs1 empty"
  "s1'" == " Norm  s1"
  "s2"  == "       st globs2 empty"
  "s2'" == " Norm  s2"
  "s3"  == "       st globs3 locs3 "
  "s3'" == " Norm  s3"
  "s4"  == "       st globs3 locs4"
  "s4'" == " Norm  s4"
  "s6'" == "(Some (StdXcpt NullPointer), s4)"
  "s7'" == "(Some (StdXcpt NullPointer), s3)"
  "s8"  == "       st globs8 locs8"
  "s8'" == " Norm  s8"
  "s9'" == "(Some (StdXcpt IndOutBound), s8)"

end
```

```
(*  Title:      Isabelle/Bali/AxExample.thy
    ID:         $AxExample.thy,v 1.6 2000/11/23 09:57:30 oheimb Exp $
    Author:     David von Oheimb
    Copyright   2000 Technische Universitaet Muenchen
*)

AxExample = AxSem + Example +

constdefs
  arr_inv :: "st ⇒ bool"
 "arr_inv ≡ λs. ∃obj a T el. globs s (Stat Base) = Some obj ∧
                              snd obj (Inl (arr, Base)) = Some (Addr a) ∧
                              heap s a = Some (Arr T #2,el)"
end
```

Bibliography

[ACR] Isabelle Attali, Denis Caromel, and Marjorie Russo. Oasis project: Java semantics. `http://www-sop.inria.fr/oasis/java/java_sem.html`.

[ACR98] Isabelle Attali, Denis Caromel, and Marjorie Russo. A formal executable semantics for java. In *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.

[Acz82] Peter Aczel. A system of proof rules for the correctness of iterative programs – some notational and organisational suggestions. Unpublished, 1982.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, 1997.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java[tm] Programming Language, Third Edition*. Addison-Wesley, 2000. `http://java.sun.com/docs/books/javaprog/`.

[AGKS99] David Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. *Proof General*, 1999.

[AL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 682–696. Springer-Verlag, 1997.

[And86] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[Apt81] Krzysztof R. Apt. Ten years of Hoare logic: A survey — part I. *ACM Trans. on Prog. Languages and Systems*, 3:431–483, 1981.

[Asp00a] David Aspinall. Proof General: A generic tool for proof development, 2000. `http://www.dcs.ed.ac.uk/home/proofgen/`.

[Asp00b] David Aspinall. Protocols for interactive e-proof. Technical Report CSE 00-009, Oregon Graduate Institute, 2000. TPHOLs 2000 Supplemental Proceedings; paper available at `http://zermelo.dcs.ed.ac.uk/~da/drafts/#eproof`.

[AZD00] Davide Ancona, Elena Zucca, and Sophia Drossopoulou. Overloading and inheritance in Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `http://www.informatik.fernuni-hagen.de/import/pi5/workshops/ecoop2000_papers.html`.

[BCM+93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, volume 18 of *ACM SIGPLAN Notices*, pages 29–46, October 1993.

[Ber00] Stefan Berghofer. Prototyping functional logic specifications. Manuscript, 2000.

[BGG+92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland/Elsevier, 1992.

[BGS95] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proc. European Conference on Object-Oriented Programming*, volume 952 of *Lect. Notes in Comp. Sci.*, pages 27–51. Springer-Verlag, 1995.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Generic Java specification. Manuscript, 1998.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[Bru93] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298. ACM Press, 1993.

[BS99] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 353–404. Springer-Verlag, 1999.

[BW98] Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Center for Computer Science, May 1998. `http://www.abo.fi/~mbuechi/publications/CompoundTypes.html`.

[BW99] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot et al., editor, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lect. Notes in Comp. Sci.*, pages 19–36. Springer-Verlag, 1999.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[CGQ98] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *OOPSLA'98 Workshop Formal Underpinnings of Java*, 1998.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[CKRW97] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In *Algebraic methodology and software technology: AMAST'97*, volume 1349 of *Lect. Notes in Comp. Sci.*, pages 75–90. Springer-Verlag, 1997.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[Coo89] William Cook. A proposal for making Eiffel type-safe. In *Proc. European Conference on Object-Oriented Programming*, pages 57–70. Cambridge University Press, 1989.

[dB99] Frank de Boer. A WP-calculus for OO. In *Foundations of Software Science and Computation Structures*, volume 1578 of *Lect. Notes in Comp. Sci.*, pages 135–149. Springer-Verlag, 1999.

[DE97a] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proc. 4th Int. Workshop Foundations of Object-Oriented Languages*, January 1997.

[DE97b] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proc. European Conference on Object-Oriented Programming*, volume 1241 of *Lect. Notes in Comp. Sci.*, pages 389–418. Springer-Verlag, 1997.

[DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 41–82. Springer-Verlag, 1999.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, 1993.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In J. Bezivin and P.-A. Muller, editors, *The Unified Modeling Language. UML'98: Beyond the Notation*, volume 1618 of *Lect. Notes in Comp. Sci.*, pages 330–348. Springer-Verlag, 1999.

[GH98] David Griffioen and Marieke Huisman. A comparison of pvs and isabelle/hol. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lect. Notes in Comp. Sci.*, pages 123–142. Springer-Verlag, 1998.

[Gib94] W. Wayt Gibbs. Trends in Computing: Software's chronic crisis. *Scientific American*, pages 86–???, September 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GM93] Michael J.C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.

[GMW79] Michael J.C. Gordon, Robin Milner, and C.P. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1979.

[Gor75] Gerald A. Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Department of Computer Science, University of Toronto, 1975.

[Gor85] Michael J.C. Gordon. HOL — a machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Gor89] Michael J.C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, volume 1783 of *Lect. Notes in Comp. Sci.*, pages 284–303. Springer-Verlag, 2000.

[HM95] Peter V. Homeier and David F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38:131–141, 1995.

[HM96] Peter V. Homeier and David F. Martin. Mechanical verification of mutually recursive procedures. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Int. Conference on Automated Deduction*, volume 1104 of *Lect. Notes in Comp. Sci.*, pages 201–215. Springer-Verlag, 1996.

[HO99] Martin Hofmann and David von Oheimb. Handling mutual recursion. Personal Communication, April 1999.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hof97] Martin Hofmann. Semantik und Verifikation. Lecture notes, in German, 1997.

[HOLa] The Isabelle/HOL library. `http://isabelle.in.tum.de/library/HOL/`.

[HOLb] HOL home page. `http://www.cl.cam.ac.uk/Research/HVG/HOL/`.

[Isa] Isabelle home page. `http://isabelle.in.tum.de/`.

[J+a] Bart Jacobs et al. Loop project. `http://www.cs.kun.nl/~bart/LOOP/`.

[J+b] Bart Jacobs et al. Project Verificard. `http://www.cs.kun.nl/VerifiCard/`.

[JBH+98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, pages 329–340, 1998.

[Jon90] Cliff B. Jones. *Systematic Program Development Using VDM.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1990.

[JP00] Bart Jacobs and Eric Poll. A logic for the Java Modeling Language JML. Technical Report CSI-R0018, CSI, 2000. `http://www.cs.kun.nl/csi/reports/info/CSI-R0018.html`.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. 4th Annual Symp. Theoretical Aspects of Computer Science*, number 247 in Lect. Notes in Comp. Sci., pages 22–39. Springer-Verlag, 1987.

[Kar98] David A. Karp. *Windows 98 Annoyances.* O'Reilly, 1998. See also `http://www.annoyances.org/`.

[Kle98] Thomas Kleymann. Hoare logic and VDM: Machine-checked soundness and completeness proofs. Ph.D. Thesis, ECS-LFCS-98-392, LFCS, 1998.

[KN00] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP2000 Workshop on Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000. `http://www4.in.tum.de/~nipkow/pubs/lbv.html`.

[Kow77] Tomasz Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.

[L+96] Jacques-L. Lions et al. Ariane 5 flight 501 failure report by the inquiry board. `http://java.sun.com/people/jag/Ariane5.html`, July 1996.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 132–145, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

[MPH99] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999. Technical Report 263, `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nip98] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[Nip99] Tobias Nipkow. *Isabelle/HOL. The Tutorial*, 1999.

[Nip00] Tobias Nipkow. Verified bytecode verifiers. Technical report, Institut für Informatik, TU München, 2000. Submitted for publication.

[NN92] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992. Revised edition (of 1999): `http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html`.

[NO98] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, 1998. `http://isabelle.in.tum.de/Bali/papers/POPL98.html`.

[NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*, pages 117–144. IOS Press, 2000. `http://isabelle.in.tum.de/Bali/papers/MOD99.html`.

[NOPK] Tobias Nipkow, David von Oheimb, Cornelia Pusch, and Gerwin Klein. Project Bali. `http://isabelle.in.tum.de/Bali/`.

[Nor98] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[Nor99] Michael Norrish. Deterministic expressions in C. In S.D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lect. Notes in Comp. Sci.*, pages 147–161. Springer-Verlag, 1999.

[NPW94] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle's logics: HOL. In *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. Up-to-date version: `http://isabelle.in.tum.de/doc/logics-HOL.pdf`.

[NW98] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.

[NYT99] The year 2000 problem. The New York Times Company, http://www10.nytimes.com/library/tech/reference/millennium-index.html, 1999.

[Ohe98] David von Oheimb. Operational semantics of Java and subject reduction, July 1998. Talk at the Colloquium "Logic in Computer Science" in Munich. Slides: http://isabelle.in.tum.de/Bali/slides/EvalTrans.ps.gz.

[Ohe99] David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1738 of *Lect. Notes in Comp. Sci.*, pages 168–180. Springer-Verlag, 1999. http://isabelle.in.tum.de/Bali/papers/FSTTCS99.html.

[Ohe00a] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. http://isabelle.in.tum.de/Bali/papers/ECOOP00.html.

[Ohe00b] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. Technical Report CSE 00-009, Oregon Graduate Institute, 2000. TPHOLs 2000 Supplemental Proceedings; paper available at http://isabelle.in.tum.de/Bali/papers/TPHOLs00.html.

[Ohe01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency: Practice and Experience*, 2001. Submitted for publication.

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1999. http://isabelle.in.tum.de/Bali/papers/Springer98.html.

[ONO00] Martin Odersky, Tobias Nipkow, and David von Oheimb. Proving type-safety of generic Java. Personal Communication, June 2000.

[OP98] David von Oheimb and Cornelia Pusch. Java — formal fundiert. In C.H. Cap, editor, *JIT'98 — Java-Informations-Tage 1998*, Informatik Aktuell, pages 77–86. Springer-Verlag, 1998. In German, http://isabelle.in.tum.de/Bali/papers/JavaDays98.html.

[OSR92] Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction — CADE-11*, volume 607 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1992.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 146–159, 1997.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau94a] Lawrence C. Paulson. Introduction to Isabelle. In *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. Up-to-date version: http://isabelle.in.tum.de/doc/intro.pdf.

[Pau94b] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994. For an up-to-date description, see http://isabelle.in.tum.de/.

[PB97] Roly Perera and Peter Bertelsen. The unofficial Java bug report. `http://www2.vo.lu/homepages/gmid/java.htm`, June 1997.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *Lect. Notes in Comp. Sci.*, pages 162–176. Springer-Verlag, 1999.

[PHT+98] Benjamin Pierce, Haruo Hosoya, David Turner, Zhaohui Luo, Philip Wadler, Kim Bruce, Sophia Drossopoulou, Vijay Saraswat, Robert O'Callahan, Matt Timmermans, Anthony Dekker, Gary T. Leavens, Luo Gang Chen, Drew Dean, Matthias Felleisen, Donald Syme, Carl Gunter, Robert Harper, and David von Oheimb. Subject reduction fails in Java. e-mail discussion on the Types Forum, archive: `http://www.cis.upenn.edu/~bcpierce/types/archives/1997-98/threads.html#00400`, June 1998.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[Pus98a] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle. Technical Report TUM-I9816, Institut für Informatik, TU München, 1998.

[Pus98b] Cornelia Pusch. Proving the soundness of a Java bytecode verifier in Isabelle/HOL. In *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lect. Notes in Comp. Sci.*, pages 89–103. Springer-Verlag, 1999.

[PVS] PVS home page. `http://pvs.csl.sri.com/`.

[Qia99] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 271–311. Springer-Verlag, 1999.

[Sch97] Thomas Schreiber. Auxiliary variables and recursive procedures. In *Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 697–711. Springer-Verlag, 1997.

[Sli96] Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 381–397. Springer-Verlag, 1996.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000. `http://www.wkap.nl/oasis.htm/257993`.

[Sun99a] Smart Card overview. Sun Microsystems, `http://java.sun.com/products/javacard/smartcards.html`, 1999.

[Sun99b] Java Card technology. Sun Microsystems, `http://java.sun.com/products/javacard/`, 1999.

[Sym97] Donald Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

[Sym99a] Donald Syme. *Declarative Theorem Proving for Operational Semantics.* PhD thesis, University of Cambridge Computer Laboratory, 1999.

[Sym99b] Donald Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 83–118. Springer-Verlag, 1999.

[TJ97] L. Robert Tolley and Diane L. Johnson. *Smart Card Handbook: Putting the World at Your Fingertips.* Smart Card International, Inc., 1997.

[Wal97] Charles Wallace. The semantics of the Java programming language: Preliminary version. Technical Report CSE-TR-355-97, University of Michigan, 1997.

[Wen99a] Markus Wenzel. *The Isabelle/Isar Reference Manual.* Institut für Informatik, TU München, 1999.

[Wen99b] Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *Lect. Notes in Comp. Sci.*, pages 167–183. Springer-Verlag, 1999.

[Wen00] Markus Wenzel. A formulation of Hoare logic suitable for Isar. `http://isabelle.in.tum.de/library/HOL/Isar_examples/Hoare.html`, 2000.

[WF94] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[WK99] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

Index