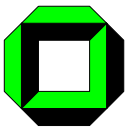

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Theoretische Informatik

Software Verification for Java 5

Diploma Thesis

by

cand. inform.
Mattias Ulbrich

Responsible Supervisor:	Prof. Dr. rer. nat. P. H. Schmitt
Supervisors:	Dipl.-Inform. Richard Bubel
	Dipl.-Inform. Steffen Schlager

Submission Date: March 28, 2007

Dank

Ich möchte mich bei meinen Betreuern Richard Bubel und Steffen Schlager für ihre stete Bereitschaft, mich bei der Erstellung dieser Arbeit zu untersützen, bedanken. Besonderen Dank möchte ich Herrn Prof. Dr. Schmitt aussprechen, der sich sehr viel Zeit genommen hat, um sich meinen Fragen zu stellen. Meinen Eltern danke ich für Ihre andauernde Unterstüzung für mich und mein Studium.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 28. März 2007

Inhalt

KeY ist ein System zur formalen Verifikation von Programmen, die in der Programmiersprache Jave geschrieben sind. Für diese Sprache wurde mit Version 5 eine Erweiterung vorgestellt, in die mehrere neue Konzepte und Konstrukte Einzug hielten. Es liegt daher nahe, zu untersuchen, wie diese Neuheiten auf KeY übertragen werden können.

Für die Untersuchung einer Anbindung von Java 5 an KeY gibt es zwei Gesichtspunkte, unter denen eine nähere Betrachtung interessant ist. Zum einen ist die Flexibilität und Anpassungsfähigkeit von KeY bzgl. neuer Strukturen zu sondieren. Dabei ist die zentral Frage, ob die Logik und der verwendete Kalkül ohne allzu großen Aufwand an die neuen Sprachkonstrukte und -konzepte angepasst werden können.

Zum anderen ist die Sprache Java 5 selbst als Untersuchungsgegenstand von Interesse. Gegebenenfalls können die Neuheiten der Sprache zusätzliche Informationen liefern, die dem Verifikationsprozess zugute kommen können. Sie können auf der anderen Seite aber auch Schwachstellen besitzen, die unter Umständen mithilfe des KeY-Werkzeugs entdeckt und umgangen werden können.

Gegenstand dieser Diplomarbeit ist die Untersuchung folgender Neuerungen der Programmiersprache Java nach der dritten Auflage der Sprachspezifikation:

1. Typsichere Aufzählungstypen (Typesafe Enumeration Types)
2. Iterationsschleifen (Enhanced For Loops)
3. Primitive Datenwerte als Objekte (Autoboxing)
4. Generischer Polymorphismus (Generic classes)

Für die Erweiterungen, die unter den ersten drei Punkten aufgeführt sind, wird eine Anbindung an das KeY-System vorgestellt. Diese Neuerungen wurde auch in einer Implementierung größtenteils umgesetzt und sind daher im System verfügbar.

Für die Generika, die gleichzeitig die tiefgreifenste Neuerung sind, konnte im Rahmen dieser Arbeit keine Implementierung angefertigt werden. Die Spracherweiterung wird eingehend vorgestellt und ein formales Gerüst vorgeschlagen, das sie auf die Logik abbildet. Für einige Probleme, die sich durch die Einführung von generischen Typen ergeben, wird gezeigt, das sie mittels eines erweiterten Kalküls gelöst werden können.

Contents

1. Introduction

1.1 Motivation

Due to the rapid technological development, growing numbers of computer systems find their way into our everyday life and into safety critical environments such as financial software, medical devices, avionics, or automotive systems. There, faulty components can cause very serious damage so that these environments require both soft- and hardware standards with minimum failure rates.

The present procedures to prevent errors in software or hardware systems are to perform a vast number of tests. Such tests, however, can only *suggest* but not *prove* the correctness of an implementation under all circumstances. Formal verification follows a fundamentally different path: The correctness of an implementation with respect to a formal specification is undeniably proven by means of formal, mathematics-related methods and verification tools.

Thanks to progress in formal methods in recent years, formal software verification is now on the verge of becoming a technique that can be widely employed for industrial purposes. The KeY system is such a tool for software verification. It is fully integrated into the design process of object-oriented software and can be used to verify software written in the Java programming language.

In 2004, version 5 of this language has been released. Java 5 involves several extensions adding new concepts and constructs to the language. The question consequently resulting from these innovations is whether KeY should be extended in order to support them. Being an approach that claims to be prepared for application in industrial contexts, the development of the KeY system must keep pace with the progress of the industrial standard. Hence, Java 5 may not be excluded from KeY and should be examined with particular emphasis on the following two aspects:

The flexibility and extensibility of KeY in general and of its logic and calculus in particular should be examined by integrating the new ideas. It ought to be analysed whether the tool is flexible enough to successfully cope with the syntactic and semantic issues introduced by Java version 5.

But it is also possible to focus on the Java language. The novelties might be capable to provide the verification process with additional knowledge and thus lead to shorter proofs. On the other hand, they could possibly have weak points, and the KeY verification environment might be powerful enough to detect and prevent typical errors introduced along with the new constructs.

Bringing Java 5 and KeY together will reveal new aspects in both parties and is therefore worth a thorough investigation.

1.2 Goal of this Work

The goal of this thesis is to examine a subset of the new features of version 5 of the Java language after the third edition of the Java Language Specification. Seven new concepts have found their way into the language, but three of them are not of relevance for formal verification and therefore are left aside.

This work covers the four constructs with impact on the formal verification process in KeY:

1. Typesafe Enumeration Types
2. Enhanced For Loops
3. Autoboxing and -unboxing
4. Generics

For some of them, namely the enumerations, enhanced for loops and parts of the boxing mechanism, implementations to integrate them into the KeY system will be provided. For the generic extension a formal framework is outlined and the problems that come along with parametric polymorphism are discussed.

1.3 Outline

The thesis is subdivided in the following chapters:

Chapter 2 introduces the KeY system, the Java Dynamic Logic and the sequent calculus used in KeY. Release 5 of the Java programming language is presented and its novelties are outlined. Related works are examined.

Chapter 3 treats the new typesafe enumeration datatypes and their integration into the KeY system. Their nature is defined formally and seized in rules for the KeY calculus.

Chapter 4 deals with the new enhanced for loop and lists rules in the calculus that transform the new statements to traditional Java. It proposes a rule that uses an invariant to prove correctness of enhanced for loops.

Chapter 5 examines the new autoboxing and -unboxing mechanisms that are used to bring primitive datatypes and reference types closer together. Rules that capture the properties defined in the Java Language Specification are proposed.

Chapter 6 covers the extension of the programming language by generic classes. This is the most far-reaching new construct amongst the innovations. The fundamentals of the logic and the calculus are augmented to deal with generics. The main problems that arise with generic classes are stated.

Chapter 7 summarises the results of chapters 3 to 6. The compatibility of the new Java and KeY is reflected and an outlook given upon what might yet be done on the topic.

2. Fundamentals

This chapter will make the reader familiar with the required fundamental concepts and definitions that are needed to understand the following chapters.

Sections 2.1 to 2.3 introduce the KeY system, its logic JavaDL and the sequent calculus that it uses.

Section 2.4 gives an outline of version 5 of the Java programming language and the novelties that have been brought into the language.

In section 2.5 examines a number of projects that deal with similar approaches to verification of Java and that might deal with the new constructs also.

2.1 Software Verification in KeY

KeY is a system for formal software verification. It has been devised to allow an approach that is fully integrated into the design and development process of object-oriented software. KeY has got interfaces that allow to use it with industry standard applications like Borland Together or the widespread developer environment Eclipse.

For specifications KeY currently supports three languages of input:

- The **Object Constraint Language** (OCL) is a part of the Unified Modelling Language (UML) and provides a way to annotate entities in the object oriented development process with formal specifications.
- The **Java Modeling Language** (JML) can be used to enrich Java sources with formal specifications and has been developed as a mean to provide 'design by contract' in the Java language.
- **JavaDL** is the internal language to which OCL and JML are translated in the KeY system. JavaDL will be introduced in extenso in the next section. Specifications can also be directly established in the language of the logic which is more powerful than OCL and JML.

KeY verifies software written in the JavaCard programming language which is a proper subset of the Java language, designed to be used for smart card applications. It does not include concurrency, reflection, and other integral parts of Java. The new constructs that came into the language Java with release 5 (see section 2.4) were mostly not brought into JavaCard as well. Amongst the seven novelties, only the generic extension explicitly is available in JavaCard, while others (like e.g. enumerated types) are explicitly excluded. But as KeY supports a superset of JavaCard already now (for instance multidimensional arrays), it can be extended by the non-JavaCard extensions without difficulty.

This is the reason why this thesis will not so often speak about JavaCard, but about Java. But one has always to keep in mind that only a subset of Java is covered by KeY.

2.2 A Dynamic Logic for JavaCard

The KeY system contains a calculus which is based on a logic especially designed for it. This is the Java Dynamic Logic, or JavaDL for short. JavaDL is a typed, multi-modal first-order logic whose modalities are induced by Java code. It has been introduced in [Bec01] and is explained extensively in [BHS07] chapter 3.

The fundamental notions of syntax and semantics (terms, formula, ...) in JavaDL have been inherited from the first order predicate logic, and the logic has been enriched by the modal operators

$$[\pi] \qquad \text{and} \qquad \langle\pi\rangle$$

that can be applied to a formula φ. They make a statement about the state after a piece of Java code π has been executed.

Throughout this chapter all necessary definitions for the JavaDL are given in comparative detail because (mainly in chapter 6) parts of the fundamental definitions have to revisited to adapt them to the needs of Java 5.

2.2.1 Syntax

2.2.1.1 Types

In the following, a logic that makes use of Java programs will be described. Expressions will be used both in the logic and Java language context. As Java is a statically typed language and the interpretation of terms in the logic shall represent objects and values that are produced during the run of a Java program, it makes sense to use a typed logic in which every term is of a defined type.

Definition 2.1 A **type system** $(\mathcal{T}, \sqsubseteq)$ is a *finite* set $\mathcal{T}$ of types with a binary relation $\sqsubseteq$ such that:

1. $\sqsubseteq$ is a reflexive partial order.
2. There is a distinct bottom element $\bot$, the "empty type", in $\mathcal{T}$.
3. There is a distinct top element $\top$, the "universal type", in $\mathcal{T}$.

4. All types must be within the empty and the universal type w.r.t. $\sqsubseteq$, i.e. $\bot \sqsubseteq T \sqsubseteq \top$ for all $T \in \mathcal{T}$.

5. The set of types is partitioned into a set $\mathcal{T}_a$ of abstract types and a set $\mathcal{T}_d$ of dynamic types, i.e. $\mathcal{T} = \mathcal{T}_a \overset{\circ}{\cup} \mathcal{T}_d$.

The type set is defined to be finite. Section 6.3.1 deals with the adaption of the type system to generic classes where the type system will outgrow this limitation.

The topology of types in $(\mathcal{T}, \sqsubseteq)$ must represent the type hierarchy of the Java programming language. Every primitive type, every class, and every interface has a corresponding type of the same name in JavaDL. Additionally there are model types which do not correspond to any Java type.

2.2.1.2 Signatures

JavaDL is based on first order predicate logic, so that the vocabulary to build terms and formulae resembles the predicate case. The symbols that can be used are pooled in a **signature**.

Definition 2.2 (Signature) A JavaDL signature $\Sigma = (\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym})$ is a collection of disjoint sets:

- a set VSym of symbols used as variables,
- a set FSym of symbols used as functions,
- a set PSym of symbols used as predicates,

Every symbol in Σ has got a type.

JavaDL is a typed logic which implies that the symbols are typed with types from the type hierarchy $(\mathcal{T}, \sqsubseteq)$.

- Each variable $v \in \mathrm{VSym}$ has got a type $T \in \mathcal{T}$.
- A function $f \in \mathrm{FSym}$ has a fixed number $n_f \geq 0$ of arguments which are types, hence, the type of a function is denoted as $T_1 \times \ldots \times T_{n_f} \to T$ with $T_i, T \in \mathcal{T}$.
- Each predicate $p \in \mathrm{PSym}$ has fixed a number $n_p \geq 0$ of arguments that are typed, so that the type of a predicate is written like $T_1 \times \ldots \times T_{n_p}$ with $T_i \in \mathcal{T}$.

KeY distinguishes between two kinds of functions and predicates: **rigid** and **non-rigid**. Every symbol in $\mathrm{FSym} \cup \mathrm{PSym}$ is either rigid or non-rigid.

The type hierarchy and the signature are fixed when a concrete problem is examined. We will therefore assume them to be appropriate and fixed from now on. Especially, an appropriate signature contains enough unused predicate and function symbols that can be requested on demand.

2.2.1.3 Terms

Terms in JavaDL are basically terms in the typed first order logic: Variable symbols and function symbols can be combined according to their arity and type. Please note that the subtype relationship $\sqsubseteq$ on the type set is involved in this definition as well. The parameters to a function or a predicate do not need to be of the exact argument type but may be of a subtype.

There is one addition to first order logic, however. Terms may be prefixed with updates, which are modal operators and are defined below.

Definition 2.3 (Term) Let $T \in \mathcal{T}$ be a type. Then the set Trm_T of terms of type T is defined as the smallest set that fulfils:

1. A variable $x : T \in \mathrm{VSym}$ is a term $x \in \mathrm{Trm}_T$.
2. If $f : T_1 \times \ldots \times T_n \to T \in \mathrm{FSym}$ is a function and $x_i \in \mathrm{Trm}_{S_i}$ with $S_i \sqsubseteq T_i$ for $1 \leq i \leq n$ then
$$f(x_1, \ldots, x_n) \in \mathrm{Trm}_T$$
3. If $t \in \mathrm{Trm}_T$ is a term and $\mathcal{U}$ an update (see definition 2.4) then $\mathcal{U}t \in \mathrm{Trm}_T$ is a term.

The set Trm of all terms is the disjoint union of all sets Trm_T:

$$\mathrm{Trm} = \bigcup_{T \in \mathcal{T}}^{\varnothing} \mathrm{Trm}_T$$

A function c with no arguments ($n = 0$) is called a constant and its application is written c instead of $c()$. Its type is then stated as a function with no parameters $c :\to T$.

As prefixes to a term (and to a formula as well) updates may be used. Like the Java program modalities introduced later, they describe a change of state. Updates can be used to fix the semantic of a term to the interpretation of another. Updates and program modalities have the same purpose though updates are conciser and more manageable in many points.

Definition 2.4 (Update) An update $\mathcal{U} \subset \mathrm{Trm} \times \mathrm{Trm}$ is a set of pairs of terms. Each pair $(l, v) \in \mathcal{U}$ has a location $l \in \mathrm{Trm}_{T_l}$ and value $v \in \mathrm{Trm}_{T_v}$ with $T_v \sqsubseteq T_l$. Both l and v may not themselves include updates also. Updates[1] are written like
$$\{l_1 := v_1 \parallel l_2 := v_2 \parallel \ldots \parallel l_k := v_k\}$$
if $\mathcal{U}$ is a finite set. The location l may contain variables but may not be a variable itself.

[1] In KeY there are also other kinds of updates (quantified, serial, ...) which have been omitted here.

A rigid function symbol refers to a function that cannot be altered. Since a location in an update is subject to a change, rigid function symbols may not appear there.

Restriction 2.5 A rigid function $f \in \text{FSym}$ may not be used as toplevel function in a location l of an update. f appears at toplevel if $l = f(x_1, \ldots, x_n)$ for some terms $x_i \in \text{Trm}$.

The definition of terms and updates require each other so that they are considered to be concurrent definitions that are only separated for the sake of readability.

2.2.1.4 Formulae

Like in first order logic, the application of a predicate to terms is the basic component for building formulae.

Definition 2.6 (Atomic formula) Let $p : T_1 \times \ldots \times T_n \in \text{PSym}$ be a predicate. Let $x_i \in \text{Trm}_{S_i}$ for $1 \leq 1 \leq n$ be terms of type $S_i \sqsubseteq T_i$.

Then the application

$$p(x_1, \ldots, x_n)$$

is an atomic formula.

A predicate p with no arguments is called a propositional variable. It is like a function with no arguments written p instead of $p()$. Like functions, predicates can only be employed with appropriately typed terms. As JavaDL is a modal logic, the set of formulae contains not only what can be stated in first order logic, but includes also modal operators.

Definition 2.7 (Formula) The set of all JavaDL formulae Fml is the smallest set that fulfils the following conditions:

1. Any atomic formula is a formula in Fml.
2. $\text{true}, \text{false} \in \text{Fml}$ are formulae.
3. If $\varphi_1, \varphi_2 \in \text{Fml}$ then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, and $\neg(\varphi_1)$ are formulae in Fml.
4. If $x \in \text{VSym}$, $\varphi \in \text{Fml}$ then $(\forall x.\varphi), (\exists x.\varphi) \in \text{Fml}$ are formulae.
5. If $\varphi \in \text{Fml}$ and π a piece of Java code (see next section), then $[\pi]\varphi \in \text{Fml}$ and $\langle\pi\rangle\varphi \in \text{Fml}$ are formulae.
6. If $\varphi \in \text{Fml}$ and $\mathcal{U}$ an update then the application $\mathcal{U}\varphi \in \text{Fml}$ is a formula.

Parentheses are used to clarify the structure of a formula.

The operators $[\pi], \langle\pi\rangle$, and $\mathcal{U}$ that can be applied to a formula are the modal operators. It remains to be defined which Java code may appear within the program modalities as program π.

2.2.1.5 Programs

The Java code that can appear within a program modality may make references to a program context that is not included in the modality itself. This is called the program and is like the type hierarchy or the signature considered part of the underlying basics of the used concrete logic.

It has been mentioned that only a subset of the Java language is allowed to be used in JavaDL: JavaCard. Some features of the Java language like concurrency, reflection, etc. cannot be handled in JavaDL while other features that are not part of JavaCard can.

Definition 2.8 (JavaDL program) A JavaDL program P is a collection of one or more JavaCard classes with the following restrictions[2]:

1. P is compile-time correct (can be compiled).
2. P does not contain inner classes.

A type system $(\mathcal{T}, \sqsubseteq)$ is compatible with a JavaDL program P if all classes that are defined in the program appear in $\mathcal{T}$ and the subtype relation $\sqsubseteq$ between them is congruent to the subtyping relation within the JavaCard language.

Definition 2.9 A JavaDL signature $\Sigma = (\mathrm{VSym}, \mathrm{FSym}, \mathrm{PSym})$ is **compatible** with a JavaDL program P if:

1. for each static field declaration "$Ty\ id$" within a class C there is a function
$$C{::}id : Ty \in \mathrm{FSym}$$
in the set of function symbols and
2. for each non-static field declaration "$Ty\ id$" within a class C there is a function
$$C{::}id : C \to Ty \in \mathrm{FSym}$$
in the set of function symbols.

The modal operators are induced by pieces of Java code; these code fragments are Java block statements, thus lists of Java statements separated by semicola. There are, however, several restrictions on these blocks:

- No return statement may appear outside a method-frame,
- no continue statement that does not apply to a loop may appear,
- only local variables that have been declared in the code may be used (within their scope), program variables that are defined in the logic as non-rigid functions in FSym may also be used, and

[2]The keybook [BHS07] demands unique identifiers which has been dropped here because it has to be dropped in chapter 6.3.2 for generic classes.

- the code may contain constructs that are beyond the Java language (method-frames, @-locations methods, etc.)[3]

It can sometimes be difficult to couch in terms which statement lists may be used within modalities and which may not. Some logic entities may be used in Java code modalities and some may not. These notions highly depend on implementations details of the KeY system.

2.2.2 Semantics

The semantic of a modal logic is usually defined by the concept of a Kripke structure. Such a structure comprises several worlds and their relationship. JavaDL is a modal logic, it makes statements about objects in different worlds – for instance before and after the execution of a program. In the context of a logic dealing with programming languages the worlds are usually called states.

The semantic of the JavaDL is stated for a fixed signature Σ and a fixed type hierarchy $(\mathcal{T}, \sqsubseteq)$, they are a priori given and adjusted to a specific problem.

2.2.2.1 Domain and Model

The set of objects about that statements are made is called the domain (or universe). Types are also relevant for semantics so that every object in the domain has got a unique type of the underlying type set $\mathcal{T}$.

Definition 2.10 (Domain) A typed domain $(\mathcal{D}, \delta)$ is a set $\mathcal{D}$ with a function $\delta : \mathcal{D} \to \mathcal{T}$ that assigns a type to every element in $\mathcal{D}$.

The elements of the domain that belong to a type $T \in \mathcal{T}$ can be collected in a set $\mathcal{D}_T$. Note that two sets $\mathcal{D}_{T_1}$ and $\mathcal{D}_{T_2}$ do not necessarily need to be disjoint.

$$\mathcal{D}_T := \{d \in \mathcal{D} : \delta(d) \sqsubseteq T\} \quad \text{for } T \in \mathcal{T}$$

It is now possible to give every term in the logic a meaning on the objects in the domain, which leads to

Definition 2.11 (Interpretation) Let $(\mathcal{D}, \delta)$ be a domain. An interpretation I "lifts" a function in the logic to a function in the domain and a predicate to a set of objects tuples in the domain:

$$I(f) : \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \to \mathcal{D}_T \quad \text{for any } f : T_1 \times \ldots \times T_n \to \mathcal{T} \in \text{FSym}$$

$$I(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_m} \qquad \text{for any } p : T_1 \times \ldots \times T_m \in \text{PSym}$$

Definition 2.12 (Model) The combination of a domain and an interpretation is called a **model** $\mathcal{M} = (\mathcal{D}, \delta, I)$.

The logic makes use of interpretations that do not provide a value for every function and predicate for all possible parameters. These models that define parts of the interpretation and leave parts of it open are called partial models.

[3] These will be introduced if and when used for the purpose of this thesis.

Definition 2.13 (Partial Model) $\mathcal{M} = (\mathcal{D}, \delta, I, D)$ is a partial model, if it is a model, that fixes the function and predicate symbols for some arguments. D is the function that assigns to every function and predicate symbols the tuples of objects at which they have been fixed:

$$\begin{aligned} f : T_1 \times \cdots \times T_n \to T \in \mathrm{FSym} \quad &\Longrightarrow \quad D(f) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \\ p : T_1 \times \cdots \times T_m \in \mathrm{PSym} \quad &\Longrightarrow \quad D(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_m} \end{aligned}$$

A partial model potentially leaves parts of the interpretation open that can be defined by a model that uses the partial model as basis. A model $\mathcal{M}' = (\mathcal{D}', \delta', I', D')$ **refines** a partial model $\mathcal{M}$ if it contains the partial model, i.e. the domain contains at least all object of underlying model (and possibly more), types them identically, and interprets the functions and predicates on them uniformly. Using $\mathbf{x}$ for an appropriately typed tuple of elements of the domain, this can be captured formally as:

$$\begin{aligned} \mathcal{D}' &\supseteq \mathcal{D} \\ \delta'\big|_{\mathcal{D}} &= \delta \\ I'(f)(\mathbf{x}) &= I(f)(\mathbf{x}) \quad \text{for } \mathbf{x} \in D(f),\ f \in \mathrm{FSym} \\ \mathbf{x} \in I'(p) &\Leftrightarrow \mathbf{x} \in I(p) \quad \text{for } \mathbf{x} \in D(p),\ p \in \mathrm{PSym} \\ D'(g) &\supseteq D(g) \text{ for any function or predicate } g \in \mathrm{FSym} \cup \mathrm{PSym} \end{aligned} \tag{2.1}$$

A logical formula is considered valid, if it is valid in all possible models. Confinements of the set of models to be considered are often axiomatised in the logic itself. Since JavaDL is used to verify software, it is a logic with a limited range of application. Thus, it makes sense to restrict the field of models taken into account to those that are interesting for Java verification.

This restriction is carried out by a partial model $\mathcal{M}_0 = (\mathcal{D}_0, \delta_0, I_0, D_0)$, the **initial model**, that defines the domain and contains predicates and functions whose interpretation is fixed. The initial model contains in particular the integral numbers and operations on them. The domain is tailored to the needs of the logic: Integers are objects in the domain as well as the two boolean truth values, and for every dynamic reference type, there are countably many objects.

$$\begin{aligned} \mathcal{T}_0 &\supset \{\mathrm{Int}, \mathrm{Nat}, \mathrm{Bool}, \mathrm{Null}\} \text{ plus the types induced by the Java program} \\ \mathcal{D}_{\mathrm{Bool}} &= \{\mathrm{tt}, \mathrm{ff}\} \\ \mathcal{D}_{\mathrm{Int}} &= \mathbb{Z} \qquad \mathcal{D}_{\mathrm{Nat}} = \mathbb{N} \\ \mathcal{D}_{\mathrm{Null}} &= \{\texttt{null}\} \\ \mathcal{D}_A &\quad \text{is a countably infinite set for every dynamic Java reference type } A \in \mathcal{T}_d. \end{aligned}$$

There are the following predefined functions and predicates in the initial model $\mathcal{M}_0$:

1. The object equality $\doteq\colon (\top \times \top) \in \mathrm{PSym}$ is a predicate which can be applied to any two terms and is interpreted as true if and only if the terms are interpreted to the very same object.

2. If $T \in \mathcal{T}\backslash\{\bot\}$ is a type, the cast operator $\big(T\big) : \top \to \mathcal{T} \in \mathrm{FSym}$ is a function of which the result is of type T and that can be applied to any object. If the argument is already adequately typed, the function behaves like the identity, it maps to an arbitrary, yet fixed element in all other cases:

$$I(\big(T\big))(x) = \begin{cases} x & \text{if } x \in \mathcal{D}_T \\ sth_T & \text{otherwise} \end{cases} \quad \text{for an arbitrary } sth_T \in \mathcal{D}_T$$

3. If $T \in \mathcal{T}$ is a type, the instanceof predicate $\sqsubseteq T : \top \in \mathrm{PSym}$ and the exact-instanceof predicate $\sqsubseteq_1 T : \top \in \mathrm{PSym}$ are defined by

$$I_0(\sqsubseteq T) = \mathcal{D}_T \qquad I_0(\sqsubseteq_1 T) = \{x \in \mathcal{D} : \delta(x) = T\}.$$

4. For each dynamic type $T \in \mathcal{T}_d$ there is a function $get_T : \mathrm{Nat} \to T$ (the so-called **repository access**) whose range $\mathrm{rng}(I_0(get_T)) = \mathcal{D}_T$ is the set of all objects of type T. This function is used to model object creation on the heap and is used in chapter 3 for enumerated types.

5. There are constants $\texttt{null} :\to \mathrm{Null}, \mathrm{TRUE} :\to \mathrm{Bool}$, and $\mathrm{FALSE} :\to \mathrm{Bool}$ in the signature with obvious interpretations. The integral numbers are available and interpreted as themselves.

6. The operations on the number and boolean types (like $<, \leq, *, +, \&, |, \dots$) are included in the signature Σ and have the expected behaviour.

There are symbols present in the signature that have a fixed meaning but that are not fixed in the initial model. The number $count_T$ of created instances of a type T is such symbol with a predefined meaning. Yet, it cannot be fixed in the initial model since the number of instances of T may change during the run of a program.

2.2.2.2 Kripke Structures

JavaDL makes use of the concept of rigidity for function and predicate symbols. A function or a predicate symbol can be defined either rigid or non-rigid.

The upcoming definitions will yield that rigid symbols are uniformly interpreted in all states that can be reached while non-rigid symbols may change their interpretations under the influence of a modal operator. That is why rigid symbols are fixed in a refinement of the initial model called the Kripke seed.

Definition 2.14 A **Kripke seed** is a partial model $\mathcal{M}_s = (\mathcal{D}_0, \delta_0, I_s, D_s)$ which refines the initial model $\mathcal{M}_0$. Every rigid function or predicate has to be completely fixed in the interpretation I_s, i.e.

$$\begin{aligned} f : T_1 \times \ldots \times T_n \to T \in \mathrm{FSym\ rigid} &\implies D_s(f) = \mathcal{D}_{T_1} \times \mathcal{D}_{T_n} \\ p : T_1 \times \ldots \times T_m \in \mathrm{PSym\ rigid} &\implies D_s(p) = \mathcal{D}_{T_1} \times \mathcal{D}_{T_m} \end{aligned}$$

It is not permitted to use a rigid function symbol as target in an update since this would provoke a change of the interpretation which would conflict with definition 2.14.

A Kripke seed describes the basis for all considered states and every refinement of it describes a state. To be able to talk about more than one state and about relations between states, several refinements have to be incorporated. The notion of the Kripke structure captures this formally:

Definition 2.15 (Kripke structure) A Kripke structure $\mathcal{K} = (\mathcal{M}_s, \mathcal{S}, \varrho)$ is composed of a Kripke seed $\mathcal{M}_s$, a set $\mathcal{S}$ of states, and a function ϱ : Program $\cup$ Updates $\rightarrow \mathcal{S} \times \mathcal{S}$ that describes the access relation between states.

The set $\mathcal{S}$ consists of all refinements of the interpretation I_s

$$\mathcal{S} = \{I \text{ Interpretation} : I \text{ refines } I_s\}$$

Each state in a Kripke structure is identified with its interpretation. Conversely every possible refinement of the seed must be present in the set of states $\mathcal{S}$. The domain, however, remains constant over all states.

Observation 2.16 (Constant domain) The domain $\mathcal{D}_0$ that has been defined for the initial model $\mathcal{M}_0$ remains the same for all states $I \in \mathcal{S}$ in all Kripke structures.

2.2.2.3 Modalities

Modalities describe accessibility relations $\varrho \subseteq \mathcal{S} \times \mathcal{S}$ between states in $\mathcal{S}$. One state can pass into another via a modality.

In the traditional (uni)modal logic, there are two modal operators $\Box$ and $\Diamond$. The formula $\Box\varphi$ is valid in a state $s \in \mathcal{S}$ if the formula φ is valid in *all* states s' with $(s, s') \in \varrho$. The formula $\Diamond\varphi$, on the other hand, is valid if there is *one* state s' with $(s, s') \in \varrho$. The very same also applies to the modalities in JavaDL, only that there is more than one accessibility relation.

The accessibility relation $\varrho(\pi)$ of a valid Java program π depends on the JavaDL program which is used as underlying framework. If the program π is executed starting in the state that is described by $I_1 \in \mathcal{S}$ and terminates normally in state $I_2 \in \mathcal{S}$, the two states are in the relation induced by $\pi : (I_1, I_2) \in \varrho(\pi)$

Java is a deterministic language so that when a program π is executed starting in a particular state s_1, it will either not terminate normally (divergence or abrupt termination) or terminate in a well-defined state s_2: there is at most one successor state for s_1 under π so that $|\{s \in \mathcal{S} : (s_1, s) \in \varrho(\pi)\}| \leq 1$.

This implies that $[\pi]\varphi$ is valid if the program π terminates normally and then φ is valid or if the program does not terminate normally. $\langle\pi\rangle\varphi$ is valid if and only if the program π terminates normally and then φ is valid. The following formula is a tautology in JavaDL for any program π:

$$\langle\pi\rangle\varphi \rightarrow [\pi]\varphi \tag{2.2}$$

The semantic of the access relation for program modalities can only be described by stating the complete semantic of the whole language. The semantic of an update, however, is easier to seize. Let $\mathcal{U} = \{f(t_1, \ldots, t_n) := t\}$ be an update with a single pair, then the successor state $I' \in \mathcal{S}$ for a state $I \in \mathcal{S}$ with $(I, I') \in \varrho(\mathcal{U})$ is given via

$$I'(g)(x_1, \ldots, x_k) = \begin{cases} val_{I,\beta}(t) & \text{for } x_i = val_{I,\beta}(t_i) \text{ and } g = f \\ I(f)(x_1, \ldots, x_k) & \text{otherwise} \end{cases}$$

The semantic of the accessibility relation of parallel updates is defined accordingly[4]. There is only one update operator because the above dualism is not possible here as updates have always exactly one successor for any state in $\mathcal{S}$.

2.2.2.4 Validity of Formulae

It remains to be defined when a formula is valid in a Kripke structure. This is done by means of a relation $\models$ that describes when a formula is valid in one a state of the structure.

JavaDL allows quantified variables so that it is necessary to define a mapping of variables to objects of the domain.

Definition 2.17 A **variable assignment** β is a function $\beta : \text{VSym} \to \mathcal{D}$ that maps variables to objects in the domain in such a manner that every variable $x{:}T \in Var$ is mapped to an adequately typed object $\beta(x) \in \mathcal{D}_T$.

The modified variable assignment $\beta^d_x(v)$ evaluates to $d \in \mathcal{D}$ if $v = x \in \text{VSym}$ and to $\beta(v)$ otherwise.

Let $\mathcal{M}$ be a Kripke structure, $I \in \mathcal{S}$ a state in this structure, and $\beta : \text{VSym} \to \mathcal{D}$ a variable assignment. Then the function $\text{val}_{I,\beta} : \text{Trm} \to \mathcal{D}$ can be used to evaluate any term in the given setup to an object of the domain.

$\text{val}_{I,\beta}(x) = \beta(x)$ for all variables $x \in \text{VSym}$
$\text{val}_{I,\beta}(f(t_1, \ldots, t_n)) = I(f)(\text{val}_{I,\beta}(t_1), ..., \text{val}_{I,\beta}(t_n))$ for terms $t_1, \ldots, t_n \in \text{Trm}$
$\text{val}_{I,\beta}(\mathcal{U}t) = \text{val}_{I',\beta}(t)$ for the unique successor state I' with $(I, I') \in \varrho(\mathcal{U})$.

The validity of a formula φ in a single state of a Kripke structure is defined by the following structurally inductive definition:

$(I,\beta) \models \text{true}$	
$(I,\beta) \not\models \text{false}$	
$(I,\beta) \models p(t_1, \ldots, t_n)$	iff $(\text{val}_{I,\beta}(t_1), ..., \text{val}_{I,\beta}(t_n)) \in I(p)$.
$(I,\beta) \models \neg\varphi$	iff $(I,\beta) \not\models \varphi$
$(I,\beta) \models \varphi \wedge \psi$	iff $(I,\beta) \models \varphi$ and $(I,\beta) \models \psi$
$(I,\beta) \models \varphi \vee \psi$	iff $(I,\beta) \models \varphi$ or $(I,\beta) \models \psi$
$(I,\beta) \models \varphi \to \psi$	iff $(I,\beta) \not\models \varphi$ or $(I,\beta) \models \psi$
$(I,\beta) \models \forall x{:}T.\ \varphi$	iff $(\mathcal{M}, \beta^d_x) \models \varphi$ for all $d \in \mathcal{D}_T$
$(I,\beta) \models \exists x{:}T.\ \varphi$	iff there is one $d \in \mathcal{D}_T$ with $(\mathcal{M}, \beta^d_x) \models \varphi$
$(I,\beta) \models \langle\pi\rangle\varphi$	iff there is an interpretation I' with $(I, I') \in \varrho(\pi)$ and $(I', \beta) \models \varphi$
$(I,\beta) \models [\pi]\varphi$	iff $(I', \beta) \models \varphi$ for all interpretations I' with $(I, I') \in \varrho(\pi)$
$(I,\beta) \models \mathcal{U}\varphi$	iff $(I', \beta) \models \varphi$ for all interpretations I' with $(I, I') \in \varrho(\mathcal{U})$

Definition 2.18 The formula φ is valid in the state I and under the variable assignment β iff $(I, \beta) \models \varphi$ can be deduced from the above. A formula φ is valid in the Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \varrho)$ if it is valid in all states $I \in \mathcal{S}$ and under all variable assignments β.

[4] KeY introduces a concept of conflict strategy that is not needed for the purposes of this document.

2.3 The Sequent Calculus

A calculus is a system of rules that can be used to deduce the validity of a formula. The formula under consideration is represented by a data structure depending on the calculus. An application of a rule transforms one such structure into another.

KeY employs a sequent calculus. The data structure upon which it operates is the 'proof tree' which is a tree whose vertices are marked with sequents (see definition 2.19).

The sequent calculus is a forward calculus. Unlike other calculi (resolution or tableaux), it does not assume the logical complement of a formula and proves its unsatisfiability but works on the formula directly. This has got the advantage that the all intermediate results remain comprehensible for a human observer who ought to be able to interact in the proof.

Definition 2.19 (Sequent) A sequent is a pair (Γ, Δ) of finite sets of JavaDL formulae without free variables. A sequent is written

$$\Gamma \vdash \Delta.$$

The left hand side Γ is called the antecedent, and the right hand side Δ the succedent of the sequent.

The sequent $\Gamma \vdash \Delta$ is equivalent to the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta,$$

which consequently implies the following meaning: If all statements in the antecedent are valid in a Kripke structure, there is (at least) one valid statement in the succedent.

In addition to the data structure, the calculus needs inference rules that transform the tree of an intermediate result into another tree. The rules for the JavaDL sequent calculus are of the form

$$\frac{P_1 \quad P_2 \quad \dots \quad P_k}{C} \quad (k \geq 0)$$

with $P_1, \dots, P_k$, and C sequents. $P_1, \dots, P_k$ are called the premisses of the rule and C is called the conclusio. A rule is sound if the validity of all premisses implies the validity of the conclusio.

The rule andRight is an example of a rule that has two premisses:

$$\mathsf{andRight}\ \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \tag{2.3}$$

Rules are often – like andRight – stated in form of a schema not only containing terms and formula but also placeholders called schema variables standing for syntactical entities (formulae, terms, program elements, ...). A rule schema represents all rules that can be derived by instantiation of its schema variables.

The schema variables Γ and Δ appear very often and are placeholder for (possible empty) sets of formulae. Γ is the schema variable for the remaining formulae on the antecedent side of a sequent, whereas Δ matches the remaining formulae on the succedent side.

2.3.1 Using the Calculus

The initial prove tree for a formula $\varphi \in \mathrm{Fml}$ is a single root node which is marked with φ.

A transformation step works on a leaf of the tree. Let this leaf be marked with a formula $\psi \in \mathrm{Fml}$ so that a rule

$$\frac{\pi_1 \cdots \pi_k}{\psi}$$

having ψ as conclusio can be applied to it. For every premiss π_i a child-node to the node under consideration is added.

There are rules called closing rules that have an empty list of premisses. Applying a closing rule on a leaf marks this node as "closed". A tree is closed if there is no leaf node that is not closed. The rule axiom is such a closing rule, its conclusio is a always valid.

$$\mathsf{axiom}\ \frac{}{\Gamma, \phi \vdash \Delta, \phi} \tag{2.4}$$

A formula φ is formally deducible in the sequent calculus if there exists a sequence of rule applications on the initial tree marked with φ that leads to a closed tree.

2.3.2 An Example

Proofs trees are usually denoted with the root as lowermost node. The derived formula, which are called **subgoals**, appear in a stack fashion on top of it.

The proof tree for a formula is built from the bottom up during the proof. An open branch is then a obligation that has not yet been proven. By applying a rule to this branch, the proof obligation is changed. When a branch is closed by a rule, the validity of the leaf sequent has been justified by this last rule without premisses.

To justify a finished proof, its closed tree has to be reads top down. The closed leaf sequents are *per se* valid because of the sound closing rules. For the other nodes the soundness of the employed rules implies their validity. In the end, the root sequent has to be valid.

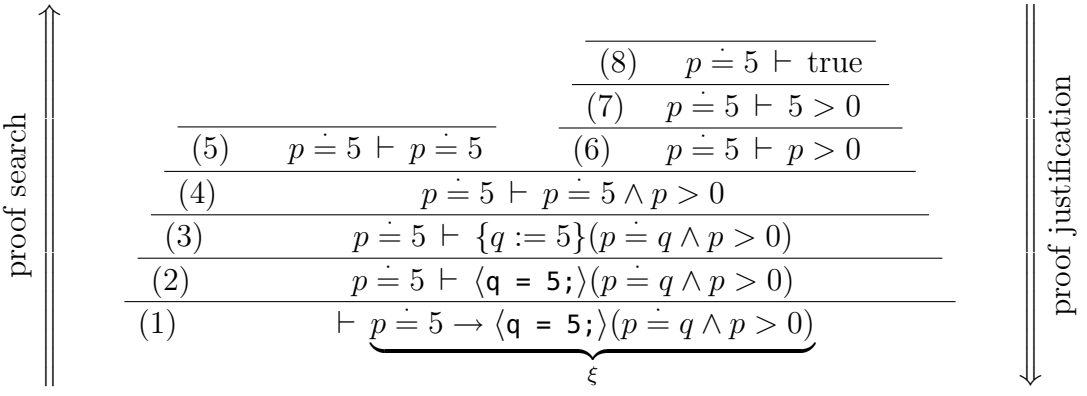

Figure 2.1: Example proof tree for the formula $p \doteq 5 \to \langle \mathtt{q = 5;} \rangle (p \doteq q \land p > 0)$

Figure 2.1 shows a sample proof for the formula $\xi = p \doteq 5 \rightarrow \langle \texttt{q = 5;} \rangle (p \doteq q \wedge p > 0)$ in which p and q are functions in the logic that take no arguments (constants). It is possible to access such functions from within a java context as variables. The formula states:

> *If $p = 5$ then after the execution of the program* `q = 5;` *p is equal to q and p is larger than zero.*

The proof begins with the root (1) which is marked with ξ. The first rule to be applied is impRight which splits up the implication and moves the premiss to the antecedent part of the sequent (2). The next rule deals with the modality in ξ and performs symbolic execution on the assignment which is then transformed into an update in (3). This update can then be applied to the formula resulting in (4). The upcoming application of andRight (cf. rule (2.3)) has got two premisses so that the tree branches into two subgoals (5) and (6). The left subgoal (5) is obviously true by axiom (cf. rule (2.4)). The right subtree needs equality $p \doteq 5$ applied to $p > 5$ first (7), followed by a literal comparison that evaluates to true in (8). The resulting sequent contains true on the succedent side, thus is trivially valid.

2.3.3 Symbolic Execution and the Active Statement

JavaDL allows program modalities which contain Java code and the calculus has to be able to perform rules also w.r.t. code within a modality to be flexible enough.

In general, it is a goal in a poof to remove all modalities within a formula by applying according rules. In order to do so, programs in modalities are executed in a symbolic way, i.e. complicated expressions and statements are broken up into simpler ones. For the basic statements, rules are given that resolve them to expressions in the logic.

For symbolic execution, rules are needed that describe the operational semantics of the Java language. Every program modality without loops and recursion can be resolved by symbolic execution to a formula without program modalities. The resulting formula may however make use of updates to express state transitions.

Since Java code can contain context information (such as try-blocks, labeled statements, etc.), it is not possible to decompose a sequence of Java statements in a modality like in classic dynamic logic like $\langle \pi \texttt{;} \nu \rangle \varphi \rightsquigarrow \langle \pi \rangle \langle \nu \rangle \varphi$. The context information would be lost. KeY does thus not provide such decomposition rules but allows application of rules to the top most statement after (a possible empty) list of opening blocks, try-statements, method-headers, etc. This statement is called the **active statement** (a.s.). The header is denoted as π and the remainder as ω, like in the following example:

$$\langle \underbrace{\texttt{label: \{ try \{}}_{\pi} \underbrace{\texttt{i = 0;}}_{\text{a.s.}} \underbrace{\texttt{if(i=0) break label; \} finally \{ i = 2; \}\}}}_{\omega} \rangle \varphi$$

Most rule schemata that are applied to programs match the active statement, so that the leading π and trailing ω are of no importance for the application of these rules.

2.3.4 Conventions

Rules are almost always stated as rule schemata. But it is often more convenient to use abbreviated forms of notation when writing downs rules. Many rules deal with very specific contexts implying that the other parts of the sequent remain unchanged. Abridged formulations are also easier to read since they concentrate on the essential.

Rules can be *rewriting* rules, i.e. they replace a formula $\psi \in \mathrm{Fml}$ at some place in the sequent by another formula $\varphi \in \mathrm{Fml}$. A rewriting rule is sound if the formulae φ and ψ are equivalent:

$$\frac{\varphi}{\psi} \text{ is an abbreviation for } \frac{\Gamma, \mathcal{U}\varphi \vdash \Delta}{\Gamma, \mathcal{U}\psi \vdash \Delta} \text{ and } \frac{\Gamma \vdash \mathcal{U}\varphi, \Delta}{\Gamma \vdash \mathcal{U}\psi, \Delta}$$

Rewriting may also be applied to terms. If $A \in \mathrm{Trm}$ and $B \in \mathrm{Trm}$ are terms equal due to some reason the rule

$$\frac{A}{B} \text{ is an abbreviation for } \frac{\{A \leftarrow B\}\varphi}{\varphi}$$

in which one occurrence of A is replaced by B.

Other rules refer to one formula $\vartheta \in \mathrm{Fml}$ in a sequent, change it and possibly add other formulae to the sequent. The schema variables $\Phi \subset \mathrm{Fml}$ and $\Psi \subset \mathrm{Fml}$ stand for sets of formulae since the application of a rule may add more than one or no formulae also.

$$\frac{\Phi \vdash \Psi}{\vartheta \vdash} \text{ is an abbreviation for } \frac{\Gamma,\ \mathcal{U}\Phi \vdash \mathcal{U}\Psi,\ \Delta}{\Gamma,\ \mathcal{U}\vartheta \vdash \Delta}$$

$$\frac{\Phi \vdash \Psi}{\vdash \vartheta} \text{ is an abbreviation for } \frac{\Gamma,\ \mathcal{U}\Phi \vdash \mathcal{U}\Psi,\ \Delta}{\Gamma \vdash \mathcal{U}\vartheta,\ \Delta}$$

2.3.5 Recoder – The Java Framework

The KeY system needs to read in Java sources and has to be able to have their syntactical structure available. There are duties that need thorough syntactical analysis of the involved programs, for instance the process that decides which method is to be invoked in the presence of overloading.

KeY does not implement these facilities, but uses the library 'Recoder' [rec] to have the source code parsed. Recoder is a Java framework for meta programming aimed to deliver an infrastructure for code analysis and transformation. It erects an abstract syntax tree for Java programs and provides it with cross referencing information. The KeY core can therefore concentrate on the logic and does not need to bother with name resolution and other analysis-related problems.

In its latest version 0.81, the Recoder framework is capable to cope with the extensions of Java 5, however, time consuming adaptions had to be performed to be have the novelties available for the KeY system as well.

2.4 Java 5

Java Standard Edition Release 5 (called Java 5) has been released on September 29, 2004 under the code name *Tiger*.

Unlike earlier release changes mainly consisting of API enhancements, the improvements for Java 5 do not only include API extensions but affect also the very language itself.

The specification [GJSB05] of Java 5 has experienced some drastic changes to the core language both in grammar and in semantics. The Java Language Specification that had not been changed since the introduction of Java 1.2 in 1998 has now been revisited and extended in many chapters.

The specification of the virtual machine – the run-time system of Java – has been left basically untouched. The syntactical extensions find their counterparts here only in additional flags and options. This unveils the topmost design principle for Java 5: downward compatibility. To allow the new version a successful start and broad acceptance, interoperability with older libraries is ensured.

This decision will be encountered in the next chapters from time to time, especially in connection with the generic extension.

2.4.1 Typesafe Enumeration Types

Traditional Java lacks the possibility to declare a datatype with a fixed number of elements. In Java 5, 'enum' datatypes fill this gap. Enumerated types can be constructed having strict properties which the verification process can take advantage of.

Chapter 3 introduces the new datatypes and describes a way to seamlessly include them into the KeY calculus.

2.4.2 Enhanced For Loops

Many programming languages have a "foreach" statement to iterate over a collection of objects. To catch up with these languages, Java 5 includes the enhanced for statement which is such a foreach loop.

While it seems that it is merely syntactic sugar, it proves to have nice properties that can be used within proofs making them significantly more efficient.

Chapter 4 deals with the new enhanced for statement, proposes rules to implement them in the KeY environment and shows their efficiency by means of an example.

2.4.3 Autoboxing and -unboxing

Generic classes are not available for primitive datatypes, so a mechanism has been invented that implicitly converts values of primitive types to references of a reference type and vice versa. Thus, primitive datatypes can also make use of the advantages that object type can benefit from.

Chapter 5 deals with autoboxing and auto-unboxing and examines how it can be implemented in the KeY calculus and outlines why this should be done.

2.4.4 Generic Classes and Generic Methods

This is the largest and most important novelty that is included in Java 5. Most of the changes in the language specification are due to the generic extension.

Chapter 6 examines the generic facility of the Java language and proposes an approach to include them into the JavaDL calculus for KeY also. Generics have proven to be a challenging topic and the logic needs some major adaptions to be able to deal with them.

2.4.5 Covariant Return Types

Type theory knows about the concept of covariant return types. If a function may be used in place of another function, their ranges do not need to coincide. It suffices if the refining function has a range that is a subtype of the original type (cf. [Pie02], ch. 15.2).

Transferred to Java methods, this means that the return type of an overwritten method in a subclass does not need to have the same return type but may have a subtype of the original type:

— Java —

```
1  class A {
2      A newInstance() {
3          return new A();
4      }
5  }
6
7  class B extends A {
8      B newInstance() {
9          return new B();
10     }
11 }
```

Java – 2.1

Before Java 5, the method `B.newInstance()` would have had to have the return type `A`. Now the more narrow type `B` is allowed. Covariant return types were initially available in Java at a very early state, but removed soon and have been reintroduced in Java 5.

The "Featherweight Java" [IPW99] core calculus has proven the type system of Java to be correct – including the covariant return types.

This feature is no closer examined in this work as it does not have any implications on the JavaDL calculus at all; it just fits seamlessly. It does provide benefit for software verification since more more type checking in a static context is performed than before.

2.4.6 Annotations

Java 5 introduces the concept of annotations for syntactical entities such as classes and class members. They can be used to tag source units of a Java program with attributes. These attributes can be read and used at compile- or run-time.

A sample attribute to attach to methods is the predefined `@Deprecated` annotation to indicate that a method has been deprecated. Other applications for annotations include attaching of meta data that do not belong into the program (copyright, version, etc.) as well as indicators for external libraries that a method or class has a certain property (`@WebService` for webservers, `@Test` for unit testing, etc.)

Annotations extend the Java language grammar by a fair amount. The grammar for the declaration of annotations deviates immensely from the usual Java syntax.

To be of use at run-time, annotations require the use of reflection methods. KeY does explicitly not support reflection, so annotation cannot have any effect on the verification with KeY.

But they might be of some use within a larger verification system: It might be possible to tag classes or methods with the results of a previous verification session. Thus, it could be possible to use a class that is delivered in byte code in a formal verification because the classes have their invariants and methods their contracts – both proven – attached as annotations.

2.4.7 Static Import

With static import, it is possible to add all static members (methods and fields) of a class to the scope of another class. This can be useful, if static operations that are collected in a class are used often in a piece of source code.

The most prominent case for a collection of operations is the class `java.lang.Math` which contains plenty of static method dealing with the evaluation of mathematical functions.

The (actually preposterous) expression

```
Math.sqrt(Math.exp(x*Math.PI) + Math.log(Math.E + Math.PI)),
```

that uses many static methods of the Math class can be written significantly shorter as

```
sqrt(exp(x*PI) + log(E+PI)),
```

if the class `lang.lang.Math` has been added to the statically imported classes.

This is – despite its clarifying influence on code – completely irrelevant to the approach to verification that KeY takes. The resolution of identifiers to variables, fields and methods to the intended entity is not the duty of the logic but of the underlying framework recoder. The logic cannot interfere with that process.

2.5 Related Work

Being both a type-safe language with a well-defined semantic and an industrial standard, Java (or related languages) are often target of projects that deal with software verification.

Bali Project Bali is a formal environment concerned with the formalisation of various aspects of the programming language Java in the theorem prover Isabelle/HOL ([vOP98]).

It captures the semantics of Java most formally and accurately and allows proofs of statements about the semantic of Java. Bali has been used to cross-verify some rules of the KeY calculus ([Wid06]).

The Bali project once had planned to formalise Generic Java also, though it seems as if this was never put into practice.

KIV KIV (formerly "Karlsruher Interaktiver Verifizierer") is a proof system which is based on algebraic specifications. Stenzel describes in [Ste05] a calculus for the Java language that also makes use of dynamic logic and sequents and looks therefore rather similar to KeY. The approach has not been extended to cover topics of Java 5.

ESC/Java2 The extended static checker for Java is not a complete formal verification tool but can be used to detect unexpected behaviour (e.g. null pointer exceptions) at compile-time. JML can be used to specify methods and classes. ESC/Java2 does not claim to detect all possible errors nor can it be used to verify arbitrary method contracts.

Currently, the static checker is limited to traditional Java but could be extended to Java 5.

Spec# Spec# ([BLS05])is an extension to the programming language C# that is widely used in Microsoft's .NET environment. It is influenced by C++ and Java – and conversely Java 5 is influenced by C#. Java and C# have many concepts in common so that a comparison is allowed.

Spec#, like the Eiffel programming language or like Java with embedded JML, allows to specify contracts for methods and invariants for classes within the source code. There exist both a dynamic checker that asserts conditions at run-time and a formal verification tool (called Boogie) that can be used to proof the specifications formally.

As the core C# language (since 2.0) supports most of the features of Java 5, namely generics, enumerated types, and foreach loops, they are also present in Spec# and the underlying calculus must be aware of them. Yet, the approach is completely different to KeY since the calculus does not operate directly on the source code but on an intermediate byte code language.

In [JMPS05] members of the Spec# research team have examined some verification procedures for special kinds of foreach loops. This special loop does not exist in Java, however.

JACK The Java Applet Correctness Kit ([BRL03]) provides an environment for verification of Java and JavaCard programs. It generates proof obligations from Java sources annotated with JML specification that are then to be proven by external provers like Simplify, Coq, or others.

JACK does not (yet) support Java 5.

Krakatoa The Krakatoa tool ([MPMU03]) allows to certify that a JML-annotated method of a Java program meets its specifications. It uses an internal language "Why" to represent the code and employs the prover Coq to prove obligations. Currently, it does not support features of Java 5.

At the moment, there seem to be no tools available and no approaches in progress that focus on the new features of Java 5 with respect to formal verification. There is some research being done on the formal fundament of generics (e.g. [IPW99], [THE+04], or [TEH05]) and its type system, but no approach to formal verification in the presence of genericity could be found.

3. Typesafe Enumeration Types

Static type safety is a paradigm that many programming languages support in order to minimise unexpected behaviour at run-time. Some of the features that were introduced with Java version 5 deal with type safety, e.g. the typesafe enumeration types.

An enumerated type is a datatype whose set of values is finite and represented by a list of unique identifiers. Many programming languages have such a construct that allows them to define a datatype consisting of a well-defined set of distinct elements. From the point of view of the implementer of a compiler, enumerated types do not pose particular difficulty as the concept of enumerated types usually will be mapped to a basic concept like primitive types or object types.

There is, however, an interesting point in enumeration types when it comes to verification. Finite sets that are uniquely represented have got properties that are easy to formulate in logic and that make them particularly interesting for verification.

3.1 Enumerated Types in Java

Prior to Java 5, storing and passing of values with a finite range had to be done with compile-time constants. If, for instance, a class `Shape` that supports multiple kinds of shapes and multiple colours had to be established, one would have solved the problem in a way similar to the following:

Java 2

```
1  class Shape {
2      // shapes
3      public static final int CIRCLE = 0;
4      public static final int ELLIPSE = 1;
5      public static final int RECTANGLE = 2;
6
7      // colours
8      public static final int RED = 0;
9      public static final int GREEN = 1;
10     public static final int BLUE = 2;
```

```
11
12      // ...
13      // Constructor
14      public Shape(int shape, int colour) {
15        // ...
16      }
17  }
```

Java 2 – 3.1

Here, enumerated values are implemented using distinct integer values. The problem is that there is no way to distinguish constants that enumerate colours from constants that enumerate shapes, all identifiers are just integer constants. In any place where an instance of an enumeration type (shape or colour) is expected, an arbitrary integer value can appear. Hence, the following constructor references are all valid in Java, but not all have the desired effect:

Java fragment

```
1   // 1) good
2   Shape shape  = new Shape(Shape.ELLIPSE, Shape.RED);
3
4   // 2) wrong order of arguments
5   Shape shape2 = new Shape(Shape.RED, Shape.ELLIPSE);
6
7   // 3) arguments are arbitrary
8   Shape shape3 = new Shape(-1, 5);
```

Java fragment – 3.2

While reference 1) has the intended semantic, with a "shape constant" `ELLIPSE` as first argument and a "colour constant" `RED` as the second, references 2) and 3) are semantically wrong. Against expectancy 2) evaluates to `Shape shape2 = new Shape(0, 1)` which has the semantic of a green circle instead of a red ellipse. This misinterpretation happens because a shape constant can be used in places where a colour constant was expected and vice versa. 3) shows that the arguments are not limited to the defined constants and that the arguments can be out of range.

This rather unsatisfying state arises because the enumeration constants are no more than compile-time integer constants and are treated as such: They have neither disjoint nor limited value ranges which one would intuitively expect. Most errors can be caught at run-time (by range-checking for instance), but there is no mean to ensure validity at compile-time already.

Joshua Bloch proposes in [Blo01] a typesafe object-oriented enumeration pattern for Java. Unfortunately, the effort to implement an enumeration type with an ordinary class type is rather high and the work tedious. His proposal is the underlying design pattern upon which the enum mechanism is based, as is explicitly mentioned in [BB04].

3.1.1 Intuitive Properties of Enumerated Types

Regardless of the semantics of a programming language, there is an intuitive semantic behind an enumerated type that can be expressed in typed first order logic.

We assume enumerations to be static, i.e. the number of the elements in the type is known a priori. There exists another, incompatible notion of enumeration types that allows an increasing (yet not decreasing) number of elements within an enumeration.

Let therefore E be an enumerated type and $\{e_1, e_2, \ldots, e_n\}$ the finite set of n constants of type E that represent all instances of the type in a unique manner. In first order logic (with equality) the proposition "There are at most n objects for the type E" can be expressed by the formula:

$$\forall x{:}E.\ (x \doteq e_1 \vee x \doteq e_2 \vee \ldots \vee x \doteq e_n). \tag{3.1}$$

But there are not only at most n objects of type E there are *exactly* n objects of E. The following formula captures the fact that the identifiers are unique. It is a representation for the proposition that "there are at least n objects of the type E":

$$e_1 \not\doteq e_2 \wedge e_1 \not\doteq e_3 \wedge \ldots \wedge e_{n-1} \not\doteq e_n \tag{3.2}$$

The two formulae 3.1 and 3.2 ensure that $\{e_1, e_2, \ldots, e_n\}$ is indeed the set of all instances of the type E.

The type system should also ensure a certain independency of E to other types. Especially, there should be no subtypes of E, for these could introduce additional instances that would destroy the postulations that have just been made.

3.1.2 Syntax and Semantics of "enum" Declarations

A complete syntax and semantics description of the enum declarations can be found in the Java Language Specification [GJSB05] §8.9 and in greater detail in the corresponding Java Specification Request [BB04].

When the Java Community Process (JCP) decided to include the enumeration mechanism into the Java language, they had two possibilities:

1. Either map enumerations internally to integers like syntactically related programming languages such as C, C++, or C# do, or
2. map enumeration types to types within the reference type hierarchy and hence provide a more complicated, yet more powerful mechanism.

The JCP voted for the second option. They claim that this was done because of the greater flexibility and greater power that can be achieved. This is certainly true as enumerations are as powerful as ordinary classes and yet the intuitive properties are preserved.

For the purposes of this document, only a subset of the actually permitted enum declarations is addressed. The approach could also be applied to the more elaborated parts, but they would go beyond the part of the language that KeY supports as they involve anonymous classes, generics and annotations.

To allow the declaration of enumerated types, a new keyword has been introduced in the Java language: "enum". It is used in place of the keyword "class" or "interface" to begin a datatype declaration. Figure 3.1 states a partial grammar for the enum declarations in Backus-Naur-Form. Terminal symbols are printed in **bold**, the nonterminal symbols are as follows:

EnumDeclaration ::=
"enum" Identifier **"{"** EnumConstants [**";"** BodyDeclaration] **"}"**

EnumConstants ::=
EnumConstant { **","** EnumConstant }

EnumConstant ::=
Identifier [**"("** ArgumentList **")"**]

Figure 3.1: A BNF grammar for enum declarations

- *Identifier*s must be valid identifiers for the Java language, naming the enum datatype or the constants. They must be unique in their context.
- *ArgumentList* is a comma-separated list of Java expressions.
- *BodyDeclaration* is a class body (member declarations) with some minor restriction (cf. JLS §8.9).

A simple example for an enum declaration is stated in listing 3.3, declaring an enum `Season` with four enum constants (l. 3) and a body declaration consisting of a static field (l. 5), a non-static field (l. 7), and a constructor (l. 9). The arguments to the enum constants (for instance 10 for `SPRING`) are used as arguments to the constructor when creating the enum constant objects.

Java 5

```
1  enum Season {
2
3      SPRING(10), SUMMER(20), AUTUMN(10), WINTER(-5);
4
5      public static final Season FAVOURITE_SEASON = SUMMER;
6
7      int typicalTemperature;
8
9      Season(int temp) {
10         this.typicalTemperature = temp;
11     }
12 }
```

Java 5 – 3.3

An enum declaration is a special kind of class declaration. It defines – like any class declaration – a reference datatype, though there are differences between a class datatype and an enum datatype:

- Once an enum datatype has been successfully initialised, there is a fixed number of instances of this type and no new instances can be created by any means.
- The existing instances are all accessible with a set of identifiers that represent all instances of the datatype in a unique manner. Thus, no object can ever be destroyed by the garbage collection process.

3.1.3 The "enum" Concept in Java

While most imperative and object-oriented languages use primitive types (integers mostly) to map enumerated types onto, the Java designers decided to embed their new concept of "enum" types into the hierarchy of reference datatypes. There is the class `java.lang.Enum` that implicitly is supertype to all enum declarations.

But there is a drawback in this decision. The property that any object of type E refers to one of the constants, which was proposed as formula (3.1), cannot be ensured. Being an ordinary reference type, it is possible for a location to hold a null-reference too, which leads to a modified, less intuitive statement

$$\forall x : E.\ (x \doteq \text{null} \vee x \doteq e_1 \vee x \doteq e_2 \vee \ldots \vee x \doteq e_n), \tag{3.3}$$

which is weaker than (3.1). It will be made clear in the following that this requires extra effort for verification as the null case has either to be excluded explicitly or to be handled separately.

Up to this point the properties (3.2) and (3.3) are a mere declaration of intent which the compiler has to implement in restrictions imposed on enumerated types. Otherwise the statements would remain on a voluntary basis and could not be used for verification.

In order to prevent the programmer from (accidentally or deliberately) creating unwanted new instances, the following restrictions have been made:

1. Explicitly referring to a constructor of an enum type yields a compile-time error.
2. The `clone()`-Method which can create new instances is overwritten and declared final in `lava.lang.Enum` and under any circumstances throws an exception.
3. No (ordinary) class declaration may extend the class `java.lang.Enum` manually and thus create a backdoor.
4. Enum datatypes may not have subtypes, they cannot be used in an "extends"-clause.
5. Creating new instances of enum datatypes via reflection is not possible.

The language guarantees with these restrictions that no other object of this class can exist but the enumeration constants.

3.2 Enum Declarations and KeY

Since enum datatypes are reference types, they behave like ordinary class types at run-time. It seems thus all natural to see enum declarations as ordinary class declarations marked as enumerations. This is in fact also the way in which the compiler treats enum declarations. The virtual machine does not differ between enum and class datatypes at run-time.

Java 2

```
1  class Season extends java.lang.Enum {
2      public static final Season SPRING = new Season(10);
3      public static final Season SUMMER = new Season(20);
4      public static final Season AUTUMN = new Season(10);
5      public static final Season WINTER = new Season(-5);
6      public static final Season FAVOURITE_SEASON = SUMMER;
7
8      int typicalTemperature;
9
10     Season(int temp) {
11         this.typicalTemperature = temp;
12     }
13 }
```

Java 2 – 3.4

Yet, this transformation alone is not sufficient. Enum classes have to be tagged as such if the verification wants to make use of the properties concerning the number and uniqueness of their instances.

Hence, when an enum declaration is found while parsing the sources of a project, the following two steps will be carried out:

1. The enum declaration is transformed into an equivalent class declaration and presented to the system.

2. Additional information on the resulting class is stored along with the class definition. This information concerns the enum constants.

3.2.1 Transformation to an Equivalent Class Declaration

The transformation[1] of an enumeration class to an ordinary class is rather straight forward. In fact, only the enum constants are affected. Each enum constant is mapped to a `public static final` class attribute and assigned a new instance which will be created at class initialisation time.

The sample class `Season` from fig. 3.3 has been transformed into the corresponding ordinary class declaration depicted in fig. 3.4. The former enum constants are now `public static final` class members which cannot be distinguished from a `public static final` class member that was originally defined as a such. In the case under consideration `Season.SPRING` and `Season.FAVOURITE_SEASON` seem to be of the same kind but are not.

It is not possible to distinguish public static class attributes that are transformed enum constants and originally defined class members. Thus, for every transformed class the original enum constants are remembered including their number and order.

[1] For the purpose of this thesis, only a skeleton translation that leaves out some implementation details is discussed

3.2.2 Type Repositories and Object Creation in KeY

JavaDL operates on a constant domain model, i.e. any interpretation must have a domain that is constant throughout all worlds of the Kripke structure under consideration. This poses the difficulty that objects that have not yet been created but might be created in an ancestor state must however be present in any state.

KeY solves this problem by using "repository access functions" which are logic functions with a predefined semantic fixed in the initial model.

Definition 3.1 A repository access function get_T for a non-abstract type $T \in \mathcal{T}_d$ is an injective, rigid JavaDL function mapping the natural numbers Nat to T:

$$get_T : \text{Nat} \to T$$

The fact that get_T is injective can be expressed with $i_1, i_2 \in \text{Nat}$ as a rule:

$$\mathsf{getInjective}\ \frac{i_1 \doteq i_2}{get_T(i_1) \doteq get_T(i_2)} \tag{3.4}$$

Definition 3.2 The number of created instances for a dynamic Type $T \in \mathcal{T}_d$ is a non-rigid JavaDL function $count_T :\to \text{Nat}$ with a value in the sort of the natural numbers.

This last definition makes sense since any Java program can within finite run-time only create a finite number of objects. The number of instances that exist of a dynamic type can vary with time so that the function $count_T$ must not be rigid. The repository access function get_T, however,< must map an index to the same object at any time and is therefore rigid.

With these two JavaDL functions, it is possible to put into a formula the fact that an object o of (dynamic) type T has been created.

$$\exists i{:}\text{Nat}.\ o \doteq get_T(i) \wedge i < count_T$$

What happens if a new object of a certain type is to be allocated? There have been $count_T$ many objects created already reachable through the access functions as $get_T(0), \ldots, g_T(count_T - 1)$. The "new" object will reside in the next free position in the repository, which is at $get_T(count_T)$. Then the number of created objects $count_T$ must be incremented by one. Finally, the initialisation of the object must be performed. This happens in the implicitly defined method `<init>` combining member initialisation and constructor code.

These are (slightly simplified) the steps that have to be taken to allocate a new instance of a dynamic type and initialise it. The rule objectCreation subsumes these steps into one rule.

$$\mathsf{objectCreation}\ \frac{\{o := get_T(count_T) \parallel count_T := count_T + 1\}\langle \texttt{o.<init>()}\rangle\varphi}{\langle \texttt{o = new T();}\rangle\varphi}$$

3.2.3 Rules for Enum Constants

If E is an enumerated type with n enum constants, the equivalent class that E is transformed into contains n static members according to section 3.2.1. These members are initialised during the initialisation of the class E. Let us have a look at how the repository access is involved when creating the instances. Consider the datatype E to be declared as follows:

JAVA 5

```
enum E { e1, e2, ..., en }
```

JAVA 5 – 3.5

As long as the class E is not initialised, there cannot be any instances of type E, so that $count_E = 0$. During the initialisation of class E (which is denoted as a call to the implicitly defined static method `<clinit>`) this numbers grows continuously till there are n instances created. Hence, once the class is successfully initialised, the equation $count_E = n$ holds. It is ensured by the compiler that the elements are created in order of appearance in the class declaration, so with the objectCreation rule applied n times the following result is achieved:

$$\frac{\begin{array}{ll}\{ & E::e_1 := get_E(0) \\ \| & E::e_2 := get_E(1) \\ \| & \dots \\ \| & E::e_n := get_E(n-1) \\ \| & count_E := n \parallel E::\texttt{<initialized>} := \text{TRUE}\}\varphi\end{array}}{\langle \pi\, \texttt{E.<clinit>();}\, \omega\rangle\varphi}$$

As soon as the class E has been initialised, it is no longer possible to create new instances of this type. Section 3.1.3 lists the means that the compiler and the environment make to ensure that this property is given.

The results that have been achieved in the update above, are therefore invariant from that time on. The references $e_1, \dots, e_n$ are final and may not be changed, and no new instances may be created. The following two rewriting rules are the intuitive results that capture this behaviour.

$$\frac{get_E(i-1)}{E::e_i} \quad (1 \le i \le n) \quad \text{and} \quad \frac{n}{count_E} \tag{3.5}$$

The case is not that easy, however, since the constant logic functions $E::e_i$ are program variables and as such non-rigid and can therefore be the target in a (user-formulated) update. The term

$$\{E::e_1 := \text{null}\}(E::e_1 \doteq \text{null}) \;\leftrightarrows\; \begin{array}{lll}\text{null} \doteq \text{null} & \rightsquigarrow & \text{true} \\ get_E(0) \doteq \text{null} & \rightsquigarrow & \text{false}\end{array}$$

can be evaluated by update simplification to true and with the rule from (3.5) to false, the calculus would become inconsistent.

The update setting the target $E::e_1$ to null in the example above is a modality that leads to a state that is not reachable by a Java program. There is no possibility in

the Java language to make the assignment like `E.e1 = null` because this attribute is declared final. To distinguish such states, which are called unreachable, from states that can be reached by a Java program, the predicate JRS (for Java Reachable State) has been introduced in the calculus (cf. [BHS07] chapter 3.3.5). It stands for a conjunction of formulae that have to hold if the state that is talked about can be reached by a Java program.

To resolve the inconsistency above, the conditions on the enum constants and the instance counter must be added to the JRS predicate so that in any reachable state $E::e_i$ and $count_E$ have the expected value:

$$\text{JRS} \longrightarrow (E::e_1 = get_E(0) \wedge \ldots \wedge E::e_n = get_E(n-1) \wedge count_E = n)$$

Thus, the rules mentioned in (3.5) may only be applied in the presence of JRS in the same state. That is why the predicate JRS must appear in the antecedent under the same update level as the formula ϕ in which the term replacement shall take place.

$$\textsf{enumConstant}\ \frac{\Gamma,\ \mathcal{U}\text{JRS},\ \mathcal{U}\{E::e_i \leftarrow get_E(i-1)\}\phi \vdash \Delta}{\Gamma,\ \mathcal{U}\text{JRS},\ \mathcal{U}\phi \vdash \Delta}$$

$$\textsf{enumConstant}'\ \frac{\Gamma,\ \mathcal{U}\text{JRS} \vdash \mathcal{U}\{E::e_i \leftarrow get_E(i-1)\}\phi,\ \Delta}{\Gamma,\ \mathcal{U}\text{JRS} \vdash \mathcal{U}\phi,\ \Delta}$$

$$\textsf{enumConstantCount}\ \frac{\Gamma,\ \mathcal{U}\text{JRS},\ \mathcal{U}\{count_E \leftarrow n\}\phi \vdash \Delta}{\Gamma,\ \mathcal{U}\text{JRS},\ \mathcal{U}\phi \vdash \Delta}$$

$$\textsf{enumConstantCount}'\ \frac{\Gamma,\ \mathcal{U}\text{JRS} \vdash \mathcal{U}\{count_E \leftarrow n\}\phi,\ \Delta}{\Gamma,\ \mathcal{U}\text{JRS} \vdash \mathcal{U}\phi,\ \Delta}$$

In the implementation for these rules there needs to be a mechanism (a meta operator outside the logic) that resolves an arbitrary enum constant $E::e_k$ to its ordinal number $k-1$, i.e. to the position within the enumeration of enum constants in the declaration. To obtain this number, it has to be stored in the additional information during the declaration transformation. To be able to use the rules that deal with the number $count_E$ of constants, this value has to be stored in the additional information as well.

3.2.4 Proving the Properties

Section 3.1.1 has introduced the following intuitive properties of an enumerated type:

$$\forall x{:}E.\ (x \doteq \text{null}\, x \doteq e_1 \vee x \doteq e_2 \vee \ldots \vee x \doteq e_n) \qquad \text{(cf. 3.3)}$$

$$e_1 \not\doteq e_2 \wedge e_1 \not\doteq e_3 \wedge \ldots \wedge e_{n-1} \not\doteq e_n \qquad \text{(cf. 3.2)}$$

It remains now to be shown that the Java enum concept fulfils these properties and that they can be proven in the KeY calculus using the rules from the previous section.

The first property is exemplarily shown for the two constants $E::e_1$ and $E::e_2$, all other inequalities follow analogously. The reachable state predicate JRS must be present in (1) to be able to apply enumConstant′. The repository access function in

(2) can be resolved using the injectivity rule getInjective (see (3.4)). The remainder is the simplification on integer numbers.

$$\dfrac{\dfrac{}{(3)\quad \text{JRS} \vdash \neg(0 \doteq 1)}}{(2)\quad \text{JRS} \vdash \neg(get_E(0) \doteq get_E(1))}$$
$$(1)\qquad \text{JRS} \vdash \neg(E{::}e_1 \doteq E{::}e_2)$$

The second property is more complicated, but the essential is that after deducing several statements about the element under consideration in (2), the problem can be reduced to a problem on the natural numbers (3) which can be proven automatically. Thanks to the repository access mechanism in KeY, problems on enum constants often dissolve in problems on their indices in the repository, which are hence on the natural numbers.

$$\dfrac{\dfrac{}{(3)\quad x \doteq get_E(c), c < n \vdash c \doteq 0, \ldots, c \doteq n-1}}{\begin{array}{c}(2)\\ \end{array}\quad \begin{array}{l}\text{JRS}, x.\texttt{<created>}, x \Subset_1 E, x \doteq \text{null} \vee x \doteq get_E(c : \text{Nat}), c < n\\ \vdash\ x \doteq \text{null}, x \doteq get_E(0), \ldots, x \doteq get_E(n-1)\end{array}}$$
$$(1)\qquad \text{JRS}, x.\texttt{<created>} \vdash x \doteq \text{null} \vee x \doteq E{::}e_1 \vee \ldots \vee x \doteq E{::}e_n$$

3.2.5 Enums in Switch Statements

The new enum datatypes have another advantage over non-enum object types: They can be used in a switch-case-statement although they are object references and not primitive values. Enumerated types will most often be used for a case distinction, so it seems natural to allow the use in switch statements. The rules dealing with these statements in the calculus have to be revised since the statement with object references is not part of the traditional language. Fortunately, the handling that KeY performs up to now covers most of the new case automatically.

To symbolically execute a switch statement in JavaDL, it is transformed into a cascade of if statements according to the following schema switchToIf. The simple expression *se* is compared to the different case-label expressions l_i using the equality operator ==. In the following "then"-statement block not only the statement list stm_i following the case-label has to be executed, but also all following statement blocks. This is because if stm_i is not quited with a break statement, the block of the next label is executed also (according to the "fallthrough" semantic of the switch statement).

$$\text{switchToIf}\ \frac{\begin{array}{l}\langle\pi\ \texttt{if(}se\ \texttt{==}\ l_1\texttt{) \{}\ stm_1\texttt{;}\ stm_2\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \texttt{else if(}se\ \texttt{==}\ l_2\texttt{) \{}\ stm_2\texttt{;}\ stm_3\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \texttt{else if(}se\ \texttt{==}\ l_3\texttt{) \{}\ stm_3\texttt{;}\ stm_4\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \ldots \\ \quad \texttt{else \{}\ stm_{def}\texttt{;\}}\ \omega\rangle\varphi\end{array}}{\begin{array}{l}\langle\pi\ \texttt{switch(}se\texttt{) \{} \\ \quad \texttt{case}\ l_1\texttt{:}\ \ stm_1\texttt{;} \\ \quad \texttt{case}\ l_2\texttt{:}\ \ stm_2\texttt{;} \\ \quad \texttt{case}\ l_3\texttt{:}\ \ stm_3\texttt{;} \\ \quad \texttt{default:}\ \ stm_{def}\texttt{;} \\ \texttt{\}}\ \omega\rangle\varphi\end{array}}$$

The rule for enum types is very much the same as the equality operator `==` has the correct semantic both on primitive types and enum constants. The only difference is the additional null reference check that has to be performed before the actual case distinction takes place due to the language specification [GJSB05] §14.11.

$$\text{EnumSwitchToIf}\ \frac{\begin{array}{l}\langle\pi\ \texttt{if(}se\ \texttt{==}\ \texttt{null)\ \{ throw new NullPointerException(); \}} \\ \quad \texttt{else if(}se\ \texttt{==}\ l_1\texttt{) \{}\ stm_1\texttt{;}\ stm_2\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \texttt{else if(}se\ \texttt{==}\ l_2\texttt{) \{}\ stm_2\texttt{;}\ stm_3\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \texttt{else if(}se\ \texttt{==}\ l_3\texttt{) \{}\ stm_3\texttt{;}\ stm_4\texttt{;}\ \ldots\ \ stm_{def}\ \texttt{\}} \\ \quad \ldots \\ \quad \texttt{else \{}\ stm_{def}\texttt{;\}}\ \omega\rangle\varphi\end{array}}{\begin{array}{l}\langle\pi\ \texttt{switch(}se\texttt{) \{} \\ \quad \texttt{case}\ l_1\texttt{:}\ \ stm_1\texttt{;} \\ \quad \texttt{case}\ l_2\texttt{:}\ \ stm_2\texttt{;} \\ \quad \texttt{case}\ l_3\texttt{:}\ \ stm_3\texttt{;} \\ \quad \texttt{default:}\ \ stm_{def}\texttt{;} \\ \texttt{\}}\ \omega\rangle\varphi\end{array}}$$

3.3 Exhaustive Switch Statements – an Example

Listing 3.6 outlines a scenario in which there is an enum declaration `Month` which contains twelve enum constants `JAN`, ..., `DEC`. There is also an ordinary class `Example` which has a non-static method `daysInMonth` that returns an integral value to be interpreted as the number of days in the month that is given as argument. This value depends on the member attribute `leapYear` because of `Month.FEB`.

Line 33 deserves to be looked at. The compiler does not notice that the case distinction within the switch statement is exhaustive and therefore the statement in line 33 cannot not be executed under any circumstances. If it is omitted, the compiler complains that the method could finish without returning a value. Though this cannot happen in reality, we have to add a statement after the case distinction that returns a value or terminates the method abnormally.

The method `daysInMonth` has got a method contract specification given in JML which states:

JAVA 5

```
1  enum Month { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
2
3
4  class Example {
5
6      boolean leapYear;
7
8
9      /*@ public normal_behavior
10       @   requires month != null;
11       @   ensures \result > 0 && \result <= 31;
12       @*/
13     int daysInMoth(Month month) {
14         switch(month) {
15             case Month.JAN:
16             case Month.MAR:
17             case Month.MAY:
18             case Month.JUL:
19             case Month.AUG:
20             case Month.OCT:
21             case Month.DEC:
22                 return 31;
23
24             case Month.APR:
25             case Month.JUN:
26             case Month.SEP:
27             case Month.NOV:
28                 return 30;
29
30             case Month.FEB:
31                 return leapYear ? 29 : 28;
32         }
33 ▷       throw new Error();
34     }
35
36 }
```

> *If the argument month is not the null reference, then the method terminates normally and the return value is an integer greater than 0 and less than or equal to 31*

Using the KeY proving instrument it is possible to prove this method contract in a proof tree with 1300 nodes and 16 branches almost completely automatically, only one manual rule application is needed.

This example shows the necessity of verification for the enum construct. The exhaustiveness of a pattern matching is not detected by the compiler, and it is thus the verification which needs to make sure that under any circumstances the argument finds a suitable match. If accidentally one case had been forgotten, the verification would detect the incompleteness and the proof would remain unclosed.

3.4 Conclusion

The enumerated datatypes are a welcome extension to the language specification from the point of view of the KeY verification system. They allow to concisely declare a datatype with a fixed number of unique elements. The very same could be achieved with an old-fashioned class declaration combined with class invariants, but the support of the concept by the language itself

a) reduces the verification workload as the invariants do not need to be proven and are guaranteed by means of the language, and

b) keeps the source code clean of such invariants and puts the intention of enumerated types more into the center.

Because of the manner in which KeY models the object creation process and the heap memory, enums lead to straight forward rewriting rules that formalise their essentials. They fit well into the KeY environment.

4. Enhanced For Loops

Java inherited its loop-control statements from the C++ programming language directly with very little changes. The while, do-while, and for loop are essentially the same as they have already been in the C programming language.

Traditional for loops are rather clumsy when used to traverse a collection or an array of entities, and many modern programming languages, such as Python or C# provide control constructs that allow to iterate collections and arrays more conveniently. These so called "for-each" loop statements permit to write code that operates on the elements of a collection in a certain order so that the implementer does not need to care about the implementation of the iteration but can concentrate on the core task.

Since the introduction of the Java Collection Framework, the typical way to iterate a collection has contained lots of iteration details that led to code that was difficult to read. When deciding to expand the syntax for Java 5, the language designers chose to add a "for-each" loop statement to overcome that deficiency, naming it the "enhanced for loop".

This chapter will examine how the new loop statement fits into the KeY calculus for traditional Java and will provide rules that symbolically execute them.

Though it seems at first glance that the enhanced for loop is no more than syntactic sugar, there are properties of these loops that simplify certain proof steps significantly in comparison to formulations with traditional statements. Such an improvement is presented in 4.3.

4.1 Syntax and Semantics of the Enhanced For Loop

The new for statement is defined and explained in the Java Language Specification [GJSB05] in paragraph 14.14.2.

Figure 4.1 shows a grammar for the enhanced for loop in Backus-Naur-Form. Here, *Expression* needs to be either an expression evaluating to an array or an expression

EnhancedForStatement ::=
    "for" "(" FormalParameter **":"** Expression ")"
        Statement

FormalParameter ::=
    Type Identifier

Figure 4.1: BNF for enhanced for loops

evaluating to a object type that implements the interface `java.lang.Iterable`. Arrays over primitive types are allowed also. *Identifier* is the variable which holds the elements one after the other and may be used within the loop body *Statement*.

Usually enhanced for loops are used in patterns like the following examples:

— Java 5 —

```
1  import java.util.*;
2
3  class EnhancedFor {
4      public static void main(String args[]) {
5
6          //
7          // Iterable
8          List list = new ArrayList();
9          list.add("a String");
10         list.add(new Integer(42));
11         for(Object item : list) {
12             System.out.println(item);
13         }
14
15         //
16         // Array
17         int[] array = {1,2,3,4};
18         for(int item : array) {
19             System.out.println(item);
20         }
21     }
22 }
```

— Java 5 – 4.1 —

In both sample cases the variable `item` is local to the statement block that belongs to the corresponding loop. The `ArrayList` object in the first case is traversed in the canonical order of the list. The order, in which items are enumerated, depends on the kind of collection that is iterated. Here `item` will hold a reference to a string in the first iteration and a reference to an `Integer` object in the second. The array is traversed in its natural order (`array[0]`, ..., `array[array.length-1]`).

The expression to be iterated does not have to be of the raw type `java.lang.Iterable` but can be of a parametrised version `java.lang.Iterable<T>` for a reference type T. Then the type reference in the formal parameter may be any supertype of T instead of `Object` in listing 4.1. But as parametrised generic types are a novelty to Java 5

also and will be dealt with separately in chapter 6, they will be omitted for the purposes of this chapter. Nonetheless, it would be possible to obtain analog results for parametrised types, too, as the applied methods do not depend on the absence of genericity.

The language specification defines the semantics of the enhanced for loop by specifying equivalent code in traditional Java. It describes the enhanced for loop as "shorthands" for the substituted traditional for loops so that the two statements – the enhanced loop and its traditional substitute – have got the same behaviour in any situation.

The replacement code which is to be stated in the following, differs significantly depending on whether the traversed expression is an `Iterable` or an array. Regardless of the nature of the iterated expression, let the enhanced for loop under consideration be according to the following template:

JAVA 5 FRAGMENT

```
l1: l2: ... ln: for( Type Id : Expression ) Statement
```

JAVA 5 FRAGMENT – 4.2

The labels $l_1, l_2, \ldots, l_n$ have to be **all** labels in front of the enhanced for loop ($n = 0$ if there are no labels in front of the loop), and the *italic* identifiers are placeholders for the appropriate syntactical units expected at this point.

4.1.1 Substitution Code for Arrays

JAVA 2 FRAGMENT

```
1 {
2     Type[] #a = Expression;
3
4   l1: l2: ... ln:
5     for (int #i = 0; #i < #a.length; #i++) {
6         Type Id = #a[ #i ] ;
7         Statement
8     }
9 }
```

JAVA 2 FRAGMENT – 4.3

The enhanced for loop for an expression of array type is semantically the same as the block statement given in listing 4.3.

1. The reference *Expression* is first copied to a freshly created local variable `#a`.

2. In a traditional for loop the array `#a` is traversed from index 0 to index `#a.length` − 1. Herein, the counter variable `#i` is freshly created.

3. In each loop iteration the variable `Id` is declared locally and assigned the i-th element of the array `#a`, and then the original loop body *Statement* is executed.

4. The labels move from before the statement to inside the block (again in front of the for statement).

4.1.2 Substitution Code for Iterators

Java 2 fragment

```
1  l1: l2: ... ln:
2    for ( Iterator #i = Expression.iterator(); #i.hasNext(); ) {
3        Type Id = #i.next();
4        Statement
5    }
```

Java 2 fragment – 4.4

If *Expression* does not evaluate to an array, the object to be iterated is of a type that implements the interface `Iterable`, thus needs to implement the method `iterator()` which returns an object implementing the interface `Iterator`. An iterator is expected to return all the objects that are contained in a collection one after the other and in a certain order that depends on the nature of the iterated object. All collections of the Java Collection Framework have been retrofitted to implement the interface `Iterable`, but own classes may implement it also.

An enhanced for loop that goes over an object implementing `Iterable` is equivalent to the code in listing 4.4. The following observations can be made:

1. The enhanced for loop is mapped to a traditional for loop.

2. A local variable `#i` is created and initialised with an iterator belonging to the iterated expression in the initialise part of the for statement. `#i` is a new identifier that does not yet appear elsewhere in the examined Java program.

3. As long as the method `hasMore()` called on the iterator `#i` returns true, the loop body is executed. The actual implementation of `hasMore()` depends on the dynamic type of the iterator. It must be guaranteed that the `next()`-method can return a sensible value if `hasNext()` evaluates to true.

4. In the loop body the identifier from the formal parameter of the enhanced loop is initialised with the next object returned by the iterator. `#i.next()` needs to have a value since `#i.hasNext()` evaluated to true. Again, it is up to the dynamic type of the iterator how `next()` is implemented.

Please note that if *Expression* has a parametrised type `Iterable<E>` for some type E, this translation looks slightly different (using `Iterator<E>` instead of `Iterator`). But as mentioned above, generics will be be kept unattended in this chapter.

In comparison to the array case, where the translation is completely independent of the type of the array, here the behaviour depends quite notably on the implementation of the iterator-object that is returned by the `Iterable.iterator()` method.

4.2 Symbolic Execution of Enhanced For Loops

Since the java language specification itself defines the semantic of the new statement by giving an equivalent statement block in traditional java, and since the virtual machine has not been changed due to the new loop type, it appears to be natural to behave accordingly on the side of verification: Express the new loop by an old one.

4.2.1 Transformation Rules

The rules that reduces enhanced for loops to traditional loops are rather straight forward as they merely are an implementation of the transformations from listings 4.3 and 4.4 which were taken from the specification directly.

There are, however, minor tweaks that have been made in order to improve integration into KeY:

1. A while-loop is used instead of a for-loop in the case that an `Iterable` object is traversed.
2. The treatment of the labels is slightly different, using an extended semantic that is illegal in pure Java.

The following rule schema is used in the calculus to symbolically execute an enhanced for loop in case that the expression is an array. It introduces a traditional for loop instead of the enhanced for construct. The names of the local program variables a and i are chosen in such a manner that no name conflicts arise.

$$\text{enhancedForArray}\ \frac{\begin{array}{l}\langle\pi\ \texttt{\{}\ ty[\,]\ a = exp\texttt{;}\\ \quad \texttt{for(int i = 0; i < } a\texttt{.length; i++) \{}\\ \quad\quad ty\ x = a[i]\texttt{;}\\ \quad\quad P\\ \quad \texttt{\}}\\ \texttt{\}}\ \omega\rangle\varphi\end{array}}{\langle\pi\ \texttt{for(}ty\ x\texttt{ : }exp\texttt{)\{}\ P\ \texttt{\}}\,\omega\rangle\varphi}$$

It is assumed by the KeY system that all programs used as input are valid Java programs. So ty and exp must be compatible, i.e. ty must be a type to which the components of exp can be cast without explicit cast operator. Hence, the resulting code is valid w.r.t. the language specification, and the correctness of the rule is given directly by the translation mentioned in the JLS.

The rule schema for the symbolic execution of an enhanced for loop with an `Iterable` object is similarly direct:

$$\text{enhancedForIterable}\ \frac{\begin{array}{l}\vdash \langle\pi\ \texttt{\{}\ \texttt{Iterator}\ it\ \texttt{=}\ exp\texttt{.iterator();}\\ \quad \texttt{while(}it\texttt{.hasNext()) \{}\\ \quad\quad ty\ x = it\texttt{.next();}\\ \quad\quad P\\ \quad \texttt{\}}\\ \texttt{\}}\ \omega\rangle\varphi\end{array}}{\vdash \langle\pi\ \texttt{for(}ty\ x\texttt{ : }exp\texttt{)\{}\ P\ \texttt{\}}\,\omega\rangle\varphi}$$

The local variable it is chosen in such a way that no name conflict will appear. As generic types are ignored here, the type schema variable ty will have to be instantiated with `Object`, as the `Iterator.next()` method returns `Object`. As soon as generics are a part of KeY, this rule can simply be adapted and type restrictions will be removed.

This rule differs a little to the original transcription since it uses a while loop instead of a for loop. But the semantics of a for loop without an update statement can be canonically expressed by a while loop so that this translation is correct also.

4.2.2 Relaxation of the Java Language Semantics

These translations are very similar to the original translations given in the language specification. Yet, there is one difference: The labels are not explicitly moved inside the newly created block, in fact, if there are any, they stay where they were – in front of the enclosing block.

The following example, for which `array` is assumed to be of type `int[]`, shows where labels in front of an enhanced for loop are located after symbolic execution.

— Java 5 fragment —

```
1 label: for(int x : array) {
2     if(x == 0)
3         continue label;
4 }
```

— Java 5 fragment – 4.5 —

symbolic execution
$\Longrightarrow$
enhancedForArray

— Java fragment —

```
1 label: {
2     int[] a = array;
3     for(int i = 0; i < a.length; i++) {
4         int x = a[i];
5         if(x == 0)
6             continue label;
7     }
8 }
```

— Java fragment – 4.6 —

The resulting code is **not** valid according to the language specification. The label `label:` in front of the opening block brace is still correct, but the reference to this label from within the continue statement is no longer correct. In §14.16 of the specification the label for a continue statement (the continue target) is restricted:

> The continue target must be a while, do, or for statement or a compile-time error occurs.

After all, `label` does not refer to a loop but to a statement block. The same phenomenon occurs also for the rule over `Iterable`s. But this little semantic unpleasantness can be encountered in the KeY calculus already now. Listings 4.7 and 4.8 show a case in which a while-loop is unrolled. First the loop condition is checked, and if it holds, the body is executed followed by the loop itself. The translation of the loop body works correctly: A new label `label2` is created and used as target for the break statement. In the remaining while loop however, the target is not relabeled and, thus, a wrong continue reference to the outer label occurs.

There are rules that make a label wander from before a loop statement to before a non-loop statement, thus creating contexts that are illegal in Java. However, this illegality does not do much harm, because the semantic of Java programs can be relaxed a little to seamlessly support this freedom of formulation.

— Java fragment ——————————

```
1 int i = 0;
2 label: while(i == 0) {
3         continue label;
4 }
```

——————— Java fragment – 4.7 —

symbolic execution
$\Longrightarrow$
unwindWhile

— Java fragment ——————————

```
1 label: if(i == 0) {
2    label2: { break label2; }
3    while(i == 0) {
4        continue label;
5    }
6 }
```

——————— Java fragment – 4.8 —

Relaxation of §14.16 A continue statement with label l attempts to transfer control to an enclosing loop lp. A loop is a statement which is either a while-, a for-, or a do-while-statement. lp then ends the current iteration and begins a new one. lp is the loop which fulfils that

a) lp encloses the continue statement,

and b) there is no loop enclosing lp in the scope of the label l.

If there is no loop that fulfills the requirements, a compile-time error occurs.

The reason why the changed semantic still is coherent is that wrong labels only appear in intermediate states. Even though the code is wrong at a time, when the enclosing loop has become the active statement, the labels will be in front of it. As KeY only works on the active statement, no harm has been done.

The problem could also be addressed by including the preceding labels into the affected rules and thereby place them into the right spot. But this is currently not possible due to the design of the syntax tree that is used by the underlying framework Recoder.

4.3 A Loop Rule with Invariants

When looking at an enhanced for loop and at its intended semantics, one gets the impression that some of these properties might be helpful to make formal statements about the loop code.

Informally spoken an enhanced for loop is a loop in which for each element in a collection of elements a piece of code is executed. If one takes into account that any collection during the run of a Java program can contain only finite many elements at a time, it seems rather natural to assume that the loop body is executed only finite many times, and that if it terminates, the loop terminates also.

Furthermore the loop's constraints such as the exit condition or the number of iterations seem to be more fixed than in an arbitrary while loop, and it seems as if statements about the new loop could be proven more easily.

4.3.1 The While Loop Invariant Rule

Loops always are the sophisticated parts of formal verification. While other statements can be "resolved" by symbolic execution, a loop may have to be repeated an unknown (or even infinite) number of times. One way to deal with loops in verification is to work with loop invariants, which are propositions that hold before and after a loop iteration.

Especially within the Hoare calculus, the partial correctness of while loops is expressed with an invariant rule. The underlying ideas of these concepts in the Hoare logic can be transferred into the Java Dynamic Logic.

4.3.1.1 A Rule for a Simple Language

For a simple while-language a rule in the JavaDL calculus that uses a loop invariant $Inv \in \text{Fml}$ would look like

$$\text{simpleInvariantRule}\ \frac{\begin{array}{ll}\Gamma \vdash \mathcal{U} Inv & \text{(Base Case)}\\ Inv, se \vdash [P]Inv & \text{(Invariant preserved)}\\ Inv, \neg se \vdash \varphi & \text{(Use Case)}\end{array}}{\Gamma \vdash \mathcal{U}[\texttt{while(}se\texttt{)\{ }P\texttt{ \}}]\varphi,\ \Delta}.$$

The conclusion that φ is valid after the execution of a while loop holds if these three premisses can be proven:

1. The invariant must hold prior to the execution of the while loop (base case).
2. The invariant must hold after each execution of the loop body P. An iteration happens only if the loop condition se holds. It may make use of the fact that the invariant holds prior to the iteration (invariant preserved).
3. Given the invariant and the fact that the loop condition is false, the loop will end. Then the postcondition φ must hold (use case).

The box modality $[\cdot]$ is used for these rules, so no statement is made about whether or not the loop terminates.

Unfortunately, the case is a little more complicated when dealing with Java programs. The reason that makes treating real Java more complex is that Java statements may end abruptly instead of normally and that the evaluation of conditions may have a) side effects and b) terminate abruptly as well.

4.3.1.2 Abrupt Termination and new Modalities

Abrupt termination is defined and explained in [GJSB05] in §14.1. Usually statements are executed in a sequential order. There are situations in which the control flow depends on the value of an expression (loops and branches), but the program is still of a sequential nature. There exist, however, cases in which a statement causes the control flow to continue at a location that is not the next statement in the sequential order. This process of control transfer is called abrupt termination while the default sequential way is called normal termination.

If there is an abrupt termination of a statement, one of the following reasons must be the case:

1. A break statement has been executed (brk). The control flow continues at the statement following the statement that was the target of the break statement.

2. A continue statement has been executed (cnt). The control flow is transferred to an enclosing loop that starts another iteration.

3. A return statement has been executed (ret). The control flow continues at the statement that is the next in the sequence after the statement that called the enclosing method.

4. An exception has been thrown (exn). The exception may be raised explicitly by a `throw` statement or implicitly, for instance by a division by zero. The control flow continues at the appropriate finally or catch block.

JavaDL does not take this into account. For the dynamic logic a program sets two states in the Kripke structure into relation if and only if a program started in the first state terminates **normally** and results in the second state. The concept of abrupt termination is unknown to KeY. Hence, from the point of view of the modal logic, the following programs both characterise the empty relation:

```
while(true);        throw new Exception();
```

Regardless of the state started in, these programs do not terminate normally. However, the first is a divergent program and the second is a program that transfers the control flow abruptly.

In order to deal with these cases, JavaDL is enriched by "indexed modalities", i.e. modalities $\langle\cdot\rangle_{rsn}$ and $[\cdot]_{rsn}$ in which *rsn* stands for one or more reasons of abrupt termination (see enumeration above). The binary relations on the states are then induced by the programs and the reason of termination. Two states are related with respect to the modality $\langle\pi\rangle_{rsn}$ if and only if π started in the first state terminates with reason *rsn* in the second state.

4.3.1.3 The While Rule for Java

With this complexity in mind, the invariant rule for the simple while loop must be revised. While in general the principles of the invariant rules are kept, there need to be other and more premisses that can deal with abrupt termination.

$$\text{whileInvariant}\ \frac{\begin{array}{l}\text{(Base Case)}\\ \text{(Invariant preserved)}\\ \text{(Abnormal condition termination)}\\ \text{(Abnormal body termination)}\\ \text{(Use Case)}\end{array}}{\Gamma \vdash \mathcal{U}[\pi\,\texttt{while(}nse\texttt{)\{ }p\texttt{ \}}\omega]\varphi,\ \Delta} \tag{4.1}$$

The invariant rule has now got five premisses which will be explained in the following. Some of them need a freshly created logic constant v of type boolean to store the result of the evaluation of the while condition *nse* which can terminate abruptly and have side effects.

(Base Case) The base case is the same as the one in the simple example: When beginning the execution of the loop, the invariant Inv must hold:

$$\Gamma \vdash \mathcal{U}Inv$$

(Invariant Preserved) The obligation to show that the invariant is preserved has become a little more complicated including the case that a `continue`-statement might be encountered and taking into account that the condition might have side effects:

$$Inv \vdash [v = nse;](v \doteq \text{TRUE} \rightarrow [p]_T Inv)$$

The body loop p may either terminate normally or terminate abruptly with a continue statement that refers to the examined while loop, hence $T = \{nrml^1, cnt\}$.

(Abnormal condition termination) If the evaluation of the loop condition does not terminate normally, the postcondition φ must nevertheless hold

$$Inv, \langle v = nse; \rangle_{exc} \text{true} \vdash \langle \pi\ v = nse;\ \omega \rangle \varphi$$

The only reason why the evaluation of $v = nse$, an assignment statement, may terminate abruptly is that a run-time exception is thrown, so that exc suffices as index to the modality.

(Abnormal body termination) If the loop body terminates abruptly, the postcondition φ must still hold. This is rather complicated to express, ensuring in the sequent's antecedents that the loop body terminates abruptly and re-executing the loop in the conclusion to trigger the abrupt condition. The set of possible reasons $AT = \{exc, ret, brk, cnt_{far}\}$ to be considered must include the case that a continue-statement refers to an outside loop, which is denoted as cnt_{far}:

$$Inv, \langle v = nse; \rangle (v \doteq \text{TRUE} \wedge \langle p \rangle_{AT} \text{true}) \vdash [\pi\ v = nse; p; \omega] \varphi$$

(Use Case) If the loop condition nse evaluates to false, the while loop ends and the postcondition must hold after the execution of the remaining program.

$$Inv \vdash [v = nse;](v \doteq \text{FALSE} \rightarrow [\pi\ \omega] \varphi)$$

The general while invariant rule grows rather complicated because of several side effects that may occur at various points and possible abrupt program terminations.

4.3.2 The Enhanced For Loop Invariant Rule for Arrays

The while invariant rule can be used to establish a similar rule for the enhanced for loop statement for arrays. Some of the proof obligations will be considerably easier to prove, as the enhanced for loop is a special case with a loop condition with helpful properties. First the while rule will be transferred to the enhanced for loop in a straight forward manner. Then this rule will be improved by the use of modifier sets.

[1] For the sake of uniformity, an index of "nrml" is used to denote the relation of normal termination.

4.3.2.1 The Classical Invariant Rule

For this section it is of great help if the iterated array expression is a simple expression that does not have any side effects and that does not terminate abruptly. Fortunately, this can be achieved by the following rule foreachNSE in which nse is a non-simple expression and v a freshly created variable that does not impose name conflicts:

$$\text{foreachNSE}\ \frac{\Gamma \vdash \langle \pi\ ty\ v = nse\texttt{; for(}ty\ x : v\texttt{) \{ }p\texttt{ \}}\omega\rangle\varphi, \Delta}{\Gamma \vdash \langle \pi\ \texttt{for(}ty\ x : nse\texttt{) \{ }p\texttt{ \}}\omega\rangle\varphi, \Delta}$$

This rule creates a new local program variable to hold the evaluated reference of the non-simple expression. If the evaluation can terminate abruptly, this is handled before the loop itself is examined.

It is therefore save to assume that the iterated expression is some simple expression se. The formula $\langle \pi\ \texttt{for(}ty\ x\ :\ se\texttt{)}\ \ p\ \ \omega\rangle\varphi$ is then equivalent to the formula

$$\{a := se\}\{i := 0\}\langle \pi\ \texttt{while(i < a.length) \{ }ty\ \ x\ \texttt{= a[i]; }p'\texttt{; i++; \}}\omega\rangle\varphi$$

in compliance with the definition of the enhanced for loop. a and i are program variables of the proper type that do not appear elsewhere in the regarded program and p' is the statement that is deduced from p by replacing every statement `continue [`*label*`];` in p with `{i++; continue [`*label*`];}`.

The general invariant rule for while loops can now be applied to this setup. However, the schema of this loop is far less general than for the general while loop presented above. The following observations can be made:

1. The program variables i and a are not accessible from within p, p', or ω.
2. The reference a points to the same location throughout the whole program as it is initially assigned a value that cannot be changed afterwards. The identifier a is not available within p.
3. The value of `a.length` remains constant throughout all iterations of the loop as a is not visible in p and not changed elsewhere. Arrays in Java never change their size once created.
4. i only holds values from 0 to a.`length` $- 1$ in increasing order.
5. If and only if se evaluates to null the loop condition $i < a$.`length` terminates abruptly. It does not have side effects.
6. The loop body p is executed at most a.`length` times. Thus, if all of these executions terminate normally, the loop terminates normally also.

All of these observations have an impact on the upcoming rule. They appear as premisses on the antecedent side of some of the generated subgoals. The last point of the enumeration is of great importance, as it allows to make a statement about the termination of the loop also – the diamond modality $\langle\cdot\rangle$ which includes termination can be used used instead of the box modality $[\cdot]$.

The invariant rule for a simple expression se reads:

$$\text{enhancedForArrayInvariant}\ \frac{\begin{array}{l}\text{(Null Case)}\\ \text{(Base Case)}\\ \text{(Abnormal body termination)}\\ \text{(Invariant preserved)}\\ \text{(Use Case)}\end{array}}{\Gamma \vdash \mathcal{U}\langle\pi\ \texttt{for(}ty\ x\texttt{:}\ se\texttt{)\{}\ p\ \texttt{\}}\omega\rangle\varphi,\ \Delta} \tag{4.2}$$

An application of the rule launches five subgoals which make use of the identifiers i and a which do not appear in the original program. They model the counter variable and the array copy that are introduced by the translation given in listing 4.3. The counter i is modelled by a program variable while the array copy a is modelled by a rigid function. Its rigidity may safely be assumed because a is assigned a value only once prior to the loop which can never be changed because the variable a is invisible from the scope of the loop body. Using a rigid function makes the rule more readable and simplifies the update application process. The variable x which appears in the original code is added to the set of program variables silently. The rules do not contain an explicit variable declaration statement.

There are three assumptions that are common to all branches but one so that they will be abbreviated by the symbol Ω_a:

$$\Omega_a := \{a \neq \text{null}, a \doteq \mathcal{U}se, a.\texttt{length} \doteq \mathcal{U}se.\texttt{length}\}$$

The term a refers to the traversed array *prior to* the loop. But the state in which the loop begins is described by the update $\mathcal{U}$ so that $\mathcal{U}se$ denotes the array at the beginning of the loop. Although a is a rigid term, $a.\texttt{length}$ is not, so that the equality of $a.\texttt{length}$ and $\mathcal{U}se.\texttt{length}$ is not implied by $a \doteq \mathcal{U}se$ and must be noted explicitly.

The branching sequents that are created by the rule enhancedForArrayInvariant can now be described:

(Null case) The whole rule can be seen as a case distinction between the two cases $se \doteq \text{null}$ and $se \neq \text{null}$ (in the context of $\mathcal{U}$). If se is null, then a new `NullPointerException` object is thrown. Nonetheless the postcondition φ needs to hold, and the program needs to terminate normally.

$$\Gamma, \mathcal{U}se \doteq \text{null} \vdash \mathcal{U}\langle\pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\varphi$$

(Base Case) While the base case for the while rule did not differ from the very first base case rule in this chapter, it does differ here. Since the enhanced for loop is translated into a while loop with preceding assignments, the base case looks as follows:

$$\Gamma, \Omega_a \vdash \mathcal{U}\{i := 0\}Inv$$

(Abrupt body termination) The loop condition $i < se.\texttt{length}$ in addition to observation 4 result in the following sequent ensuring that if the body of the

original loop terminates abruptly, the whole program terminates normally and that the postcondition holds then, too.

$$\begin{array}{ll} & Inv, \Omega_a, i \geq 0, i < a.\texttt{length}, \{x := a[i]\}\langle p\rangle_{AT}\,\text{true} \\ \vdash & \{x := a[i]\}\langle \pi\, p\, \omega\rangle\varphi \end{array}$$

In this particular case the set of reasons of termination to be considered is $AT = \{brk, exc, ret, cnt_{far}\}$. Though a local continue is a reason for abrupt termination, it may not be included here as it leads to another iteration of the loop rather than to an abortion and is addressed by the upcoming branch.

(Invariant preserved) under the application of the loop body.

It does not suffice to consider the execution of the original body loop alone. Before the loop body is executed, the item variable x must be set to the current array element $a[i]$. Due to the nature of a as a rigid function, this must happen in an update and cannot be performed within a Java modality.

After the loop body the counter increment statement `i++` follows. The increment may not be appended to p sequentially (as in $\langle p\texttt{; i++}\rangle Inv$) since this would not cover the case that p terminates abruptly with a continue-reason in which case the increment has to be executed nonetheless.

The case of a continue that belongs to the regarded loop must also be taken into consideration because it leads to another loop iteration as well. So the modes of termination to be considered here are $T = \{nrml, cnt\}$.

Like in the previous sequent the constants a and $a.\texttt{length}$ are assumed to hold the values they held before the loop started (Ω_a).

$$\begin{array}{ll} & Inv, i \geq 0, i < a.\texttt{length}, \Omega_a \\ \vdash & \{x := a[i]\}\langle p\rangle_T\{i := i+1\}Inv \end{array}$$

(Use case) The use case may exert the invariant and the fact that the loop condition does not hold. Though the latter would only imply that $\neg(i < a.\texttt{length})$, the stronger formula $i \doteq a.\texttt{length}$ may be used because of the extra knowledge about the loop.

$$Inv, i \doteq a.\texttt{length}, \Omega_a \vdash \langle \pi\ \omega\rangle\varphi$$

Ω_a still needs to be included, although a will not appear in $\langle \pi\ \omega\rangle\varphi$. The invariant Inv may make use both of a and i.

The enhanced for loop has got properties that simplify the rules in some points: the loop condition ($i < a.length$) cannot fail for instance. Nonetheless the sequents that appear in the enhanced for rule contain more formulae than the original ones in the while rule. The reason for this is that the for-loop is a specialised application of a general rule. All this specialisation (like the variable initialisation $a = \mathcal{U}se$) is additional knowledge that is added on the antecedent side of the sequent, thus does not create new proof branches but creates new potentials to close upcoming goals.

4.3.2.2 The Invariant Rule with Modifier Sets

The rule enhancedForArrayInvariant is of help if applied to a detached for-loop without or with little context, but becomes unhandy if the loop belongs to a larger context. Consider a case in which the loop appears after several variables have been initialised and with several preconditions assumed. Then the information about this is mirrored in the logic within the formulae in Γ and Δ and within the update $\mathcal{U}$.

The invariant Inv of the loop must be strong enough to allow to deduce the post condition φ from it. Yet, the (Use case) has no access to the original formulae in Γ, Δ or to the update $\mathcal{U}$ so that the invariant needs to capture their meaning also – although they are already present in the original sequent. This results in increased verification work, as many properties that are already proven have to be checked once again.

Therefore, the KeY system approaches the problem slightly differently. Instead of coding what *does not* change in an invariant, it keeps the original context (Γ, Δ, and $\mathcal{U}$) and declares what possibly *does* change and therefore has to be dropped. In case of a loop, that implies that also the (Use case) can apply some formulae or updates from the original context (Γ, Δ, and $\mathcal{U}$), but not all of them.

The declaration of locations that might change is performed with so called **anonymising updates** that operate on **modifier sets**. A formal definition of modifier sets and updates is given in the KeY-book [BHS07] in chapter 3.7.

Definition 4.1 (Modifier set) Given a program π a modifier set $M \subseteq \text{Trm}$ is a superset of the set of locations (ground terms) that π can change during a run.

The modifier sets is independent of the state in which π is started in and must include all locations that might be changed in any run of π. Locations within the modifier set of a loop body can be changed in one iteration and may therefore not hold the same value in the next. Statements about the original value may be wrong for this lately assigned value.

This is why an update is introduced which "clears" the value of a location by setting it to an arbitrary value, namely a freshly created skolem constant.

Definition 4.2 (Anonymising update) Let M be a modifier set. Then an anonymising update $\mathcal{V}(M)$ with respect to M is an update that sets any term $t = f(t_1, ..., t_n) \in M$ to a term that do not yet appear anywhere.

$$f(t_1, ..., t_n) \in M \implies \{f(t_1, ..., t_n) := f'(t_1, ..., t_n)\} \in \mathcal{V}(M)$$

for a new rigid function f' which has the same signature as f.

If the modifier set cannot be or is not given, a general anonymising update $\mathcal{V}(*)$ is used that sets all terms that appear in the context to freshly created terms.

The branches of the invariant rule for enhanced for loops that was introduced in the last section may now employ the improvement of anonymising updates. Not much changes for the (Null case) and the (Base case) since they already exerted the original context $\Gamma, \Delta, \mathcal{U}$. But for the (Invariant preserved), (Abrupt body termination), and

(Use case) the introduction of $\mathcal{V} := \mathcal{V}(M)$ allows the use of the original context (more precisely of the parts that are not made unavailable by $\mathcal{V}$).

The invariant rule enhancedForArrayInvariant (4.2) can now be revised to use modifier sets and anonymising updates:

$$\text{enhancedForArrayInvariantModSet}\ \frac{\begin{array}{l}\text{(Null Case)}\\ \text{(Base Case)}\\ \text{(Abnormal body termination)}\\ \text{(Invariant preserved)}\\ \text{(Use Case)}\end{array}}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{for(}\, ty\, x\texttt{:}\, se\texttt{)\{}\ p\ \texttt{\}}\,\omega\rangle\varphi,\ \Delta} \tag{4.3}$$

with

- se a simple expression whose result is an array,
- $\Omega_a := a \neq \text{null}, a \doteq \mathcal{U}se, a.\texttt{length} \doteq \mathcal{U}se.\texttt{length}$ like above,
- a a freshly created rigid constant function of the same type as se,
- x a program variable (thus a non-rigid constant) of type ty,
- i a freshly created non-rigid constant function of integer type, and
- M a modifier set for the loop body p and $\mathcal{V} := \mathcal{V}(M)$ the corresponding anonymising update.

and the premisses

(Null case)
$\Gamma,\ \mathcal{U}se \doteq \text{null} \vdash \mathcal{U}\langle \pi\ \texttt{throw new NullPointerException();}\,\omega\rangle\varphi,\ \Delta$

(Base Case)
$\Gamma,\ \Omega_a \vdash \mathcal{U}\{i := 0\}Inv$

(Abrupt body termination)
$\Gamma,\ \Omega_a,\ \mathcal{U}\mathcal{V}Inv,\ (\mathcal{V}i) \geq 0,\ (\mathcal{V}i) < a.\texttt{length},$
$\mathcal{U}\mathcal{V}\{x := a[i]\}\langle p\rangle_{AT}\,\text{true}$
$\vdash\ \mathcal{U}\mathcal{V}\{x := a[i]\}\langle \pi\, p\, \omega\rangle\varphi, \Delta$

(Invariant preserved)
$\Gamma,\ \mathcal{U}\mathcal{V}Inv,\ (\mathcal{V}i) \geq 0,\ (\mathcal{V}i) < a.\texttt{length},\ \Omega_a$
$\vdash\ \mathcal{U}\mathcal{V}\{x := a[i]\}\langle p\rangle_T\{i := i+1\}Inv,\ \Delta$

(Use case)
$\Gamma,\ \mathcal{U}\mathcal{V}Inv,\ (\mathcal{V}i) \doteq a.\texttt{length},\ \Omega_a \vdash \langle \pi\ \omega\rangle\varphi,\ \Delta$

In the branches (Abrupt body termination), (Invariant preserved), and (Use case) the counter i is subject to the anonymising update $\mathcal{V}$, too. The variable i is not visible from within the program p, so there is no need for it to be in the modifier set M of $\mathcal{V}$. So why do we still apply the update to it? The variable i does not need to be in M, but it may be (e.g. if $\mathcal{V} = \mathcal{V}(*)$ is the general anonymising update) and in that case i will appear in the context of $\mathcal{V}$ on the right hand side and, thus, must appear updated on the left hand side also.

4.3.3 No Invariant Rule for Iterators

Since the results for the enhanced for loop for arrays are quite promising, one expects similar results for the loop with iterator objects.

Unfortunately, this is not the case.

There are mainly two reason that prevent applying the ideas of the invariant rule for arrays to loops with iterators also:

1. The evaluation of the loop condition and the assignment before the loop body may have side effects,
2. quite contra-intuitively the enhanced for loop does not guarantee a finite runtime in case of its application to an `Iterable` object.

The loop condition is a call to the method `hasNext()` of the iterator that has been created implicitly. The case that this method fails, must be covered by a rule, as well as the case that the initial assignment `ty x = it.next()` (see listing 4.4) fails.

The general while rule has to deal with similar effects also so that this would not be an argument against an invariant rule. But the second point is far more severe: The invariant rule for the enhanced for loop is superior to the general while rule in one point: it guarantees termination. An enhanced for loop that traverses an array cannot diverge, if the loop body does not diverge. An enhanced for loop that traverses an `Iterable` object can, however, under certain circumstances, diverge.

Listing A.1 in the appendix depicts the implementation of a linearly-linked list `Chain`, which supports only two operations: 1) append an object to the chain, or 2) iterate the list with a `ChainIterator`, a class that is declared in this context also.

Each chain has got a reference to a chain (the tail) and a reference to the object at this position (the head). The `append` method appends an object to the end of the list by calling `append` on the tail or by creating a single-element-chain if the current position is the end. The `ChainIterator` performs the traversal of the chain by managing a reference to a chain and by advancing this pointer if needed (i.e. when `next()` is called).

The class `ChainProblem` defines the method `main` in which a Chain is created and then traversed. In “Loop 1” the array is iterated with an enhanced for statement and the contained two elements are printed to the console.

In “Loop 2” the array is traversed once more, but this time, after the current element has been printed out, a new element is appended to the end of the list. This way the list grows continuously, and though at any time the list is finite and the iterator position advances in each step, the loop will never finish. It diverges.

One might utter that it is the job of the Iterator to detect such modification during traversal and to raise an exception. But this is not part of the API specification for the interface `java.util.Iterator`. Yet, the reference implementations for the traversal of the collections in the collection framework of SUN’s Java detect “concurrent modification” and throw a corresponding exception. This, however, is not *required* for the implementation of an iterator. The class `ChainIterator` is a sane implementation that does support appending during the traversal of a collection. This may be a desired feature for some reason, yet, it is lethal if one desires convergence of the enhanced for loop.

4.3.4 Implementation Issues

The rule enhancedForArrayInvariantModSet that was presented in section 4.3.2.2 has been implemented in the KeY system. Due to technical and implementation details, the rule that has been brought into the KeY system is not the presented one. They differ in the following points:

- The counter i that indicates the current position within the array is implemented using a quantified variable rather than by means of a constant non-rigid function.
- There are no indexed modalities $\langle\cdot\rangle_{rsn}$. Instead a transformation is applied to the program that records every abrupt termination and yields a normal termination instead. The reason of the termination is then available as a set of predicates on the logic level and can be utilised to state formulae equivalent to the formulae that use indexed modalities.
- The implemented rule launches only four instead of five subgoals since without indexed modalities the normal and the abrupt termination can be combined.

4.3.5 No Parallelisable Semantic

Gedell and Hähnle have published a way to resolve loops automatically into parallel updates in [GH05]. Their approach aims at parallelisable loops whose iterations could be performed in parallel and do not depend on each other. These loops can be accurately represented by an update.

Parallel loops often arise when the elements of an array are initialised. Loops that deal with independent initialisation assignments are very often 'for loops' so that it seems possible that similar effects could be observed for certain applications of the enhanced for loop.

A parallelisable initialisation loop follows in general the pattern

$$\texttt{for(int i = 0; i < a.length; i++) a[i] = e(i)}, \tag{4.4}$$

used to assign the evaluation of an expression $e(i)$ to each position $a[i]$ of the array a. If all $e(i)$ are expressions that do not rely on a common state and do not make use of previously assigned $a[i]$, all assignments could also be performed in parallel in one step and be captured in a parallel update.

Unfortunately, the enhanced for statement cannot be used to produce loops like (4.4) because there is no counter variable i present that could be employed to address independent locations. Listing 4.9 demonstrates a situation in which an enhanced for loop is utilised to initialise the field `value` in all elements of an array `array`. This seems to be a rather special and seldom application. Transformation to a conventional for loop allows the methods of [GH05] to be applied so that these cases can be parallelised as well. But this does not justify installing extra rules for enhanced for loops and parallel initialisation.

Java 5 fragment

```
1 for(Cl cl : array) {
2   cl.value = expression;
3 }
```

Java 5 fragment – 4.9

4.4 The Minimum in an Array – an Example

The invariant rule for enhanced for loops can be used to prove statements with modalities that contain corresponding loops significantly more efficiently than the corresponding rule for while loops.

JAVA 5

```
1  class Minimum {
2
3      static int minimum(int[] array) {
4          int m = array[0];
5          for(int x : array) {
6              if(x < m)
7                  m = x;
8          }
9      }
10 }
```

JAVA 5 – 4.10

Consider the method `minimum` in listing 4.10 that is designed to compute the minimum of an array of integers. As an example, it is to be proven that the method computes the minimum element of the array given as argument.

The two formulae ϕ and ψ are employed to express this intention in JavaDL. ϕ predicates that the value of the variable *min* is less than or equal to all elements in the array *array*. The formula ψ is an invariant which makes sure that the elements of the array after the method invocation are still the same as before the call. To formulate this, the rigid function $old : \text{Int} \to \text{Int}$ models the unchanged values.

$$\psi = \forall i.\ array[i] = old(i) \tag{4.5}$$

$$\phi = \forall i.\ ((0 \leq i \wedge i < array.\texttt{length}) \to min \leq array[i]) \tag{4.6}$$

The following sequent is the specification of the method `minimum` and ensures in the antecedent that the array is not null and that its length is valid.

$$\begin{array}{rl} & array \neq \text{null}, array.\texttt{length} \geq 0 \\ \vdash & \psi \to \langle \texttt{int min = Minimum.minimum(array)} \rangle (\psi \wedge \phi) \end{array} \tag{4.7}$$

This sequent can be proven using the invariant rule for the enhanced for loop or using the transformation to a while loop and then the appropriate while loop rule. In the latter case, termination has to be proven separately. Both alternatives lead to a quasi automatic poof. The invariant rule has to be instantiated manually, but the following proof requires only a very few further manual instantiations (γ-contexts).

An invariant that can be employed to proof the contract is the formula

$$\forall k.\ ((0 \leq k \wedge k < I) \to m \leq ar[k])$$

which states that m (the variable within the method) holds a value that is less than or equal to the first I elements of the array ar. This is the partial result that can

be obtained after I iterations of the loop. As soon as the method has returned and $I = ar.\texttt{length}$ and $min = m$ the postcondition ϕ can be concluded trivially.

In the present example the invariant rule would therefore be instantiated with the following terms for the open schema variables.

$$\begin{aligned} a &\leftarrow array \\ Inv &\leftarrow \forall k.\ ((0 \leq k \wedge k < I) \rightarrow m \leq ar[k]) \\ M &\leftarrow \{m\} \\ i &\leftarrow I \end{aligned}$$

In fig. 4.2 the two proofs are compared. The rule for the enhanced for loop only needs roughly a third(!) of the steps that the general rule needs. This originates from the additional assumptions that the enhanced for rule may make because of the extra knowledge that one has got about the loop condition and the counter variable.

	enh. for	**while**
Number of nodes in the proof tree	374	1053
Number of branches in the proof tree	8	21
Number of additional manual instantiations	2	3

Figure 4.2: Comparison of the enhanced for rule and the while rule

4.5 Conclusion

Enhanced for loops are not necessary for formal verification. The Java programming language does not gain any new expressivity not present in the traditional language. But the new loop statement has a right to exist. It helps making code clearer and more readable and avoids programming errors that arose because of the clumsy syntax of the traditional for loop.

For the case of iterated arrays, the invariant rule for while loops has been adapted to enhanced for loops. The adjusted rule has given evidence to lead to significantly shorter proofs. The main advantage of this rule is that is gets the termination of the loop for granted since an enhanced for loop terminates if every iteration of its loop body terminates.

A similar invariant rule for the case of an iteration over a collection could not be established since the construct cannot guarantee termination there even if each loop iteration terminates.

5. Autoboxing and -unboxing

Java's type system knows two kinds of datatypes: primitive types and reference types, which are treated fundamentally differently. While for a primitive type a location contains the stored *value* itself, a location for a reference type contains a *reference* (or pointer) to an object's data.

All reference types have a common supertype `java.lang.Object` and can, therefore, be treated uniformly. Primitive types on the other hand are not subtypes of `java.lang.Object` and must always be handled separately because they hold values instead of references.

Hence, it is not possible to add for instance a value of integer type to a `java.util.List` collection object which can only store reference objects. Also, the new generic feature (cf. chapter 6) can only be applied to reference types so that a list of primitive integer values (such as `List<int>`) cannot be created.

Fortunately, the Java language contains a set of classes that wrap ("box") primitive values so that such a boxing object can be used instead of the primitive value. The discrimination of primitive and reference types can thus be partly overcome. But writing code with lots of such boxing situations is tiring and makes code unreadable. This is why in Java 5 these conversions happen implicitly and primitive elements can be used like reference objects.

5.1 The Boxing Mechanism in Java

Each primitive datatype has a corresponding "boxing" reference type that is a class and that captures one value of the primitive type. Such boxing objects are created with an appropriate parameter to a constructor and the stored value can be obtained by calling a query method on the object. Instances of boxing classes are immutable, i.e. the boxed value cannot be changed after creation. This, in fact, captures the behaviour of values best as the values themselves are also immutable units. Fig 5.1 lists the primitive datatypes and their corresponding reference types including the method name of the method that has to be used to retrieve the boxed data.

Primitive type	Reference type†	Query method
`boolean`	`Boolean`	`booleanValue()`
`byte`	`Byte`	`byteValue()`
`short`	`Short`	`shortValue()`
`int`	`Integer`	`intValue()`
`long`	`Long`	`longValue()`
`char`	`Character`	`charValue()`
`float`	`Float`	`floatValue()`
`double`	`Double`	`doubleValue()`

† All these classes are in the package `java.lang`, so the package prefix is omitted for the sake of readability.

Figure 5.1: Primitive types and their corresponding reference types.

The grammar of the Java language is not affected by the introduction of autoboxing and auto-unboxing. No new grammatical structures are introduced, the changes purely affect the semantics.

5.1.1 Introducing Example

Listing 5.1 shows a small example with which the boxing problem is illustrated. The class `Grades` models a mapping between names (of pupils) and grades, which are integer values. For faster access the values are stored in a hash map that uses the names as hash keys.

Java 2

```
1  class Grades {
2
3     HashMap grades = new HashMap();
4
5     public void setGrade(String name, int grade) {
6        grades.put(name, new Integer(grade));
7     }
8
9     public void addGrade(String name, int update) {
10       int grade = ((Integer)grades.get(name)).intValue();
11       setGrade(name, grade + update);
12    }
13
14    public boolean hasPassed(String name) {
15       int grade = ((Integer)grades.get(name)).intValue();
16       return grade > 50;
17    }
18 }
```

Java 2 – 5.1

A `HashMap` is a class from the Java Collection Framework that stores data for keys. But both key and data are restricted to reference objects and cannot be used for

primitive types. With traditional Java the integer values have to be manually converted to Integer objects by creating new objects of class `Integer`, and have to be unboxed to regain access to the stored value again. These explicit boxing and unboxing expressions are marked with frames in listing 5.1.

The very same functionality can be obtained with class `Grades5` from listing 5.2 that employs the newly introduced autoboxing and -unboxing functionality. It reads far more comprehensible, and if generics (see chapter 6) were used, even the remaining two casts in lines 10 and 15 would become obsolete.

Java 5

```
1  class Grades5 {
2
3     HashMap grades = new HashMap();
4
5     public void setGrade(String name, int grade) {
6        grades.put(name, grade);
7     }
8
9     public void addGrade(String name, int update) {
10       int grade = (Integer)grades.get(name);
11       setGrade(name, grade + update);
12    }
13
14    public boolean hasPassed(String name) {
15       int grade = (Integer)grades.get(name);
16       return grade > 50;
17    }
18 }
```

Java 5 – 5.2

But what happens if a name unknown to the HashMap is passed as argument to the method `hasPassed`? A `NullPointerException` will be thrown in line 15. That is due to the unboxing mechanism and could not have appeared in traditional Java. Autoboxing – better auto-unboxing – bears a new potential of unexpected exceptions. One always has to keep in mind that autoboxing and -unboxing happens implicitly and that it might lead to abrupt termination.

5.1.2 How Boxing is Performed

The Java Language Specification [GJSB05] explains the concept of boxing and unboxing in chapter 5.1 by the introduction of two new conversions.

The **Boxing Conversions** converts an expression which has a primitive datatype to an expression of the corresponding reference type such that at run-time the following condition holds: If p is the value of an expression with primitive type, the result of the boxing conversion of p is an object r of the corresponding reference type such that the content retrieval query on r results in p.

Let p for instance be an integer value then the proposition r.`intValue()` $= p$ holds.

The **Unboxing Conversion** converts an expression that has one of the reference types in fig. 5.1 to the corresponding primitive type such that at run-time the following proposition holds: If r is an object of a boxing reference type then the result

of the unboxing conversion of r is a value p of the corresponding primitive type such that p is equal to the value returned by the content retrieval query applied to r.

If for instance r is an object of type Integer, then the proposition r.`intValue()` $= p$ holds. A `NullPointerException` is thrown if r is `null`.

5.1.3 Where Autoboxing is Applied

Java is a statically typed language, every expression has a type that can be computed at compile-time on the basis of its context. There are, however, situations in which an expression of a type different to the static type is needed to fit into the surrounding context. If the type of an expression is changed, the language specification speaks about conversions, and unboxing and boxing are two of them.

The situations in which conversions can occur are called conversion contexts. The language description knows five of them, of which four are relevant for boxing:

Assignment Conversions If an assignment statement (`lhs = rhs`) assigns a reference object (rhs) to a location (lhs) of a primitive value, the reference object is subject to unboxing conversion.

If a primitive value is assigned to a location of a reference object, the expression (lhs) has to undergo boxing conversion first.

This also covers variable initialisation, so that for instance `Integer obj = 3;` will box the integer literal to an Integer object.

Method Invocation Conversion If the formal parameter of a method or of a constructor has a primitive type but the expression used as argument to the method has a reference type, the expression is subject to unboxing conversion.

And the way round: If the formal parameter is of a reference type and the expression used as argument of a primitive type then the expression is boxed to comply with the constraints.

Note that the possibility of the implicit boxing and unboxing conversions before a call to a method has implications on the process that decides which method (or constructor) to choose from a set of overloaded methods with the same name. The matching algorithm that chooses the method to call has to be extended to allow boxing and unboxing conversions.

Casting Conversion If an expression contains an explicit cast operator already, the operand may require a boxing or unboxing conversion to fulfill before the actual cast.

Thus, the explicit cast expression `(Object)3` is performed by two conversions: First a boxing conversion `(Integer)3`, and then a reference conversion (widening the reference type) from the reference type `Integer` to `Object`, like in `(Object)((Integer)3)`.

Numeric Promotion Numeric promotion converts the types of the operands of a numeric operator so that an operation can be performed upon them. If expressions of non-primitive types are involved as arguments to arithmetic or numeric operators, they have to undergo unboxing conversion first.

This allows for instance to add Integer reference objects resulting in a primitive integer value: `new Integer(3) + new Integer(4)` results in the value 7 which is a value of the primitive type `int`.

Also the expression `intObj++`, though looking strange at first glance, is allowed, if `intObj` is a reference to an object of type Integer. `intObj` is subject to unboxing, then the value 1 is added to the result. And this result again is subject to boxing conversion, of which the result is assigned to the variable `intObj` again.

The boxing and unboxing conversion does not only appear in these situations listed in the specification as conversion contexts, but also in a other contexts mentioned elsewhere in the language definition.

Control statement argument conversion This affects the for, while, do-while, and switch statement. If an expression used as the controlling expression in one of these statements has a reference type instead of a primitive type, unboxing conversion is applied to the argument.

This allows for instance to use a reference of type Boolean as loop condition in a while statement. It also permits to use the switch statement with an Integer object instead of a primitive int value.

5.2 Autoboxing and Formal Verification

The autoboxing and -unboxing facilities do not simplify the verification process. On the contrary, the additionally inserted implicit operations that have to be performed even extend the workload for verification.

These implicit conversions are nevertheless an important issue to be considered for formal correctness as they have potential side-effects which are worth to be examined in the verification process. Unexpected exceptions can appear if references to null are subject to an unboxing conversion, unknown behaviour is the case if two boxed primitive values are compared, etc.

If autoboxing and -unboxing is well described on the level of the calculus, it can help to identify and eliminate errors both in the specification and implementation.

Consider the definition of the method `meth` in listing 5.3. This example raises a `NullPointerException` at line 2 if `n==null` and at line 3 if `m==null`. Both statements (comparison and increment) cannot cause null pointer errors in traditional Java (like array accesses, object field or method accesses could). Hence, the reason for exceptions are not as obvious if conversions are made implicitly.

Java 5 fragment

```
1 void meth(Integer n, Integer m) {
2    if(n > 0) {
3       m++;
4    }
5 }
```

JAVA 5 FRAGMENT – 5.3

That is why it is important to include this new Java feature into formal verification: To make sure that the error potential imported along with the convenience is minimised.

Throughout this chapter, a way to capture boxing in the JavaDL calculus will be proposed. Most of the arguments stated in the following will deal with the pair (int, Integer) of corresponding types, some with (boolean, Boolean). But the very same mechanisms can be applied to the other primitive datatypes in a very analogical manner.

5.2.1 Boxing outside the Java Context

It is tempting to try to implement boxing that is not limited to code within modalities but may be lifted into the context of JavaDL. However, the following JavaDL proof makes clear that this must not happen:

$$(1)\ \frac{(2)\ \dfrac{(3)\ \dfrac{}{\vdash \texttt{(Integer)}1000 \doteq \texttt{(Integer)}1000}}{\vdash \{o := \texttt{(Integer)}1000 \parallel p := \texttt{(Integer)}1000\}(o \doteq p)}}{\vdash \langle \texttt{Object o = (Integer)1000; Object p = (Integer)1000;}\rangle (o \doteq p)}$$

If in (1) the explicitly mentioned boxing conversions are not performed, the term is pulled outside into the logic unchanged and placed into an update by the assignment rule. This update is then in (2) applied to the postcondition $o \doteq p$ which results in an equality of two identical terms which is always true.

But this is semantically wrong as o and p may refer to different objects which both contain the value 1000. The current reference implementation of Java does create two instances here, so that o and p **are** different objects.

This is a strong indication why boxing should be resolved within the Java context and not be lifted into the logic. Expressions that are structurally equal in Java do not need to result in equal objects in the logic.

5.2.2 Boxing Conversion Contexts

The task of handling boxing and unboxing in KeY will be subdivided into two steps:

1. Contexts in which boxing or unboxing has to take place are identified and made explicit. This section deals with the detection and preparation of boxing contexts.

2. Boxing and unboxing conversions are performed in the previously marked contexts. The next section will cover the rules for the actual boxing and unboxing.

Rules that refer to statements in modalities always match the active statement, the first statement that follows a (possibly empty) list of opening blocks, labels, try-headers, etc. It suffices thus to state rules that handle the boxing conversion contexts for the active statement.

Once a boxing context has been spotted, an explicit cast that describes the conversion is inserted. Hence, if a boxing or unboxing conversion is to be performed in the active statement, it can be recognised easily in the following because it has been tagged by an explicit cast. Autoboxing contexts are therefore transformed into explicit casting contexts.

If an expression that appears within the active statement is composed of one or more operations, these non-simple expressions are pulled out. A new variable is declared and the expression is assigned to it during symbolic execution. This is why it suffices to consider cases which match the schema

$$loc = expression$$

in which *loc* is a location (a left-hand-side) and *expression* is an expression with one operation at most.

The set of rules schemata below is used to make the boxing and unboxing contexts explicit. It contains the following scheme variables:

loc	A location (left-hand-side) that has type int
Loc	A location that has type Integer
exp	An expression of type int
Exp	An expression of type Integer
sth	An arbitrary expression
Type	A type that is a super-type of Integer, but not Integer (`Integer` $\sqsubsetneq$ *Type*)
type	A type that int can be cast to without explicit cast. This range differs from primitive type to type.
BoolExp	an expression of type Boolean
stm	a statement
op	a binary operator that takes integral arguments (`+,-,*,/,%,>>,&`,...)
unaryop	an unary operator that takes an integer argument (`-,+,~`,...)

Assignment context rules

$$\text{explicitBoxingIntAssign}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ Loc\texttt{=(Integer)(}exp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ Loc\texttt{=}exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingIntAssign}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ loc\texttt{=(int)(}Exp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\texttt{=}Exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

Casting context rules

$$\text{explicitBoxingIntCast}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ Loc\texttt{=(}Type\texttt{)(Integer)(}exp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ Loc\texttt{=(}Type\texttt{)}exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingIntCast}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ loc\texttt{=(}type\texttt{)(int)(}Exp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\texttt{=(}type\texttt{)}Exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

Numeric promotion context rules

$$\text{explicitUnboxingIntBinaryOpLeft}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ \texttt{((int)(}Exp\texttt{))}\ \ op\ \ sth\texttt{;}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ Exp\ \ op\ \ sth\texttt{;}\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingIntBinaryOpRight}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ sth\ \ op\ \ \texttt{((int)(}Exp\texttt{))}\texttt{;}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ sth\ \ op\ \ Exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingIntUnaryOp}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ Loc\ \texttt{=}\ unaryop\texttt{((int)(}Exp\texttt{))}\texttt{;}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ loc\ \texttt{=}\ unaryop\ \ Exp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

Statement argument context rules

$$\text{explicitUnboxingIf}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{if((boolean)(}BoolExp\texttt{))}\ \ stm\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{if(}BoolExp\texttt{)}\ \ stm\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingWhile}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{while((boolean)(}BoolExp\texttt{))}\ \ stm\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{while(}BoolExp\texttt{)}\ \ stm\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingDoWhile}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{do}\ \ stm\ \ \texttt{while((boolean)(}BoolExp\texttt{))}\texttt{;}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{do}\ \ stm\ \ \texttt{while(}BoolExp\texttt{)}\texttt{;}\ \omega\rangle\varphi, \Delta}$$

$$\text{explicitUnboxingSwitch}\ \frac{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{switch((int)(}Exp\texttt{))}\ \ cases\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{switch(}Exp\texttt{)}\ \ cases\ \omega\rangle\varphi, \Delta}$$

There have been no rules given for the method invocation context, which is after section 5.1.3 also a context boxing or unboxing may appear in. The reason for this is that it is not possible to deduce the method to be invoked within the calculus. This is done by the underlying framework Recoder ([rec]) that provides the abstract syntax tree and other information on the source like the resolution of the method to be invoked. The framework is aware of overloaded or overwritten methods and will choose the right alternative for the method call. There is a special rule methodCall that resolves the method and which is described on page 135ff in the key book [BHS07].

Amongst over steps, the rule methodCall pulls out the arguments to the method and assigns them to freshly created program variables that have the exact type of the formal parameter of the method. Let in the following example `C.m(int v)` be a static method, then the application of the rule creates a new variable `arg1` and assigns the actual parameter to `arg1`.

$$(1)\ \frac{\vdash \langle \texttt{int arg1 = new Integer(1); C.m(arg1)@C}\rangle\varphi}{\vdash \langle \texttt{C.m(new Integer(1))}\rangle\varphi}$$

Thus, the method invocation context in which an unboxing would have taken place is automatically transformed into an assignment context which is covered by rules given in the list above, the unboxing can be made explicit.

5.2.3 Rules for Autoboxing and -unboxing

After the first step all boxing and unboxing conversion contexts have been identified and explicit cast operations have been inserted where necessary. Since an explicit cast is an operation, it will be pulled out if it is a subexpression in a composed expression or an argument to a statement. It is assigned to a new variable during symbolic execution whenever a simple expression is needed. Thus, it is sufficient to consider cases matching the following scheme:

$$loc = (type)expression$$

with *loc* a location, *type* a Java type, and *expression* an expression. If *type* is a primitive type and the type of *expression* is a reference type, an unboxing conversion must be performed. If *type* is a reference type and the type of *expression* is a primitive type, then boxing is to be applied. In any other case it is neither a boxing nor an unboxing context.

The **unboxing** conversion in the casting conversion context can be captured by a rule in the JavaDL calculus in a rather straight forward manner. For the primitive type `int`, this would be:

$$\text{intUnboxing}\ \frac{\Gamma \vdash \langle \pi\ \texttt{v=(}expression\texttt{).intValue();}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \langle \pi\ \texttt{v=(int)}expression\texttt{;}\ \omega\rangle\varphi, \Delta}$$

In section 5.1.2 the unboxing has been defined to have exactly this property: "If r is an object of a boxing reference type then the result of the unboxing conversion of r is a value p such that p is equal to the value returned by the content retrieval query applied to r."

The rules for the other primitive types can be obtained analogically by replacing the call to `intValue` by the call to the appropriate content retrieval method from fig. 5.1.

The **boxing** conversion, however, cannot be put into rules in the same straight forward manner as the unboxing conversion. A first attempt to cover the boxing conversion is the rule intBoxNew in which v is a variable of an appropriate reference type and *iexp* a simple expression of type `int`:

$$\text{intBoxingNew}\ \frac{\Gamma \vdash \langle \pi\ v\ \texttt{= new Integer(}iexp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \langle \pi\ v\ \texttt{= (Integer)}iexp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

This rule creates a new Integer object with the integer value in question for each appearance of a boxing operation. Unfortunately, this is not in accordance with the language specification. The JLS even requires that for certain ranges in certain primitive types (e.g. the interval [-128, 127] of integer values) two boxing conversions boxing the same primitive value return the same object. It is explicitly left open how a compiler behaves outside these ranges.

Consequently a boxing context cannot be simply mapped to the creation of a new boxed object, a more complex setup has to be used. For the rule to appear next, knowledge about the boxing classes is needed. An extract of the class `java.lang.Integer` with an outline of the implementation is listed in appendix A.2: The value of interest

is stored in the field `value` of which the content can be obtained by calling the query method `intValue()`. Please note that this is the class used internally by KeY. As long as the behaviour remains the same, a Java environment may use a completely different implementation. The class's nature could also be described with constraints (in JML or OCL for instance) rather than in Java code.

When a boxing conversion is to be performed on an integer value *iexp*, what is known about the result *obj* of the conversion is:

1. *obj* is a reference to a valid object of type `Integer`,
2. *obj* is not a null reference,
3. *obj* boxes the value *iexp*, so *obj*`.intValue()` must return *iexp* and *obj*`.value` $\doteq$ *iexp*,
4. the number of created Integer objects may have increased by one (if a new Object is created) or may still remain the same, and
5. the class `Integer` will be initialised before the conversion if it has not been initialised yet.

Let's elide the static initialisation of the boxing classes here because dealing with them would require a detailed knowledge about the implementation of the class `Integer` which is not given in general.

With v like above, *iexp* a simple expression of type integer, *obj* a freshly created program variable of type `Integer`, and c a freshly created rigid constant of type integer, the rule intBoxing captures the observations from above:

$$\text{intBoxing}\ \frac{\begin{array}{l}\Gamma, c \doteq 0 \lor c \doteq 1, \mathcal{U}\neg(obj \doteq \text{null}) \\ \vdash \\ \mathcal{U}\{obj\texttt{.value} := iexp \parallel obj\texttt{.<created>} := \text{TRUE} \parallel \\ \quad count_{\texttt{Integer}} := count_{\texttt{Integer}} + c\}\langle \pi\ v\ \texttt{=}\ obj\texttt{;}\ \omega\rangle\varphi, \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle \pi\ v\ \texttt{=}\ \texttt{(Integer)}iexp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

The count function $count_{\texttt{Integer}}$ holds the number of objects that have already been created for the type `Integer` and the implicit identifier `<created>` indicates, that *obj* is valid object in this context.

The fact that some primitive values are always mapped to the same object is not covered by this rule. Though this is a property that programs should not rely upon, a rule that covers this also can be given. To fully capture the boxing intention the accurate rule adds a premiss introducing a pool of integers $IntegerCache : int \to Integer$ which is a rigid function defined no further. However, if boxing is performed on the same value $p{:}int$ twice, both boxing objects will be equal to IntegerCache(p) and thus equality of both objects can be deduced if $p \in [-127, 128]$.

$$\text{intBoxingAccurate}\ \frac{\begin{array}{l}\Gamma, c \doteq 0 \lor c \doteq 1, \mathcal{U}\neg(obj \doteq \text{null})\\ \mathcal{U}((iexp \geq -128 \land iexp \leq 127) \rightarrow obj \doteq \text{IntegerCache}(iexp))\\ \vdash\\ \mathcal{U}\{obj\texttt{.value} := iexp \parallel obj\texttt{.<created>} := \text{TRUE} \parallel\\ \quad count_{\texttt{Integer}} := count_{\texttt{Integer}} + c\}\langle \pi\ \omega\rangle\varphi, \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle \pi\, v\ \texttt{=}\ \texttt{(Integer)}iexp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

There is another approach to the boxing conversion which uses the given implementation of the wrapping classes and uses methods that are designed to construct boxing objects. While this is highly dependent on the implementation that is used, it models its behaviour most accurately as this is how the compiler performs the conversion. But in order to apply such a reduction, the concrete implementation of the class "Integer" must be on hand and must be brought into the verification process.

The API of the class Integer contains (since Java 5) a method `valueOf(int value)` which is used by the compiler to resolve boxing conversions. The rule intBoxingValueOf makes use of this method to resolve all boxing conversions with a call to this method.

$$\text{intBoxingValueOf}\ \frac{\Gamma \vdash \langle \pi\, v\ \texttt{=}\ \texttt{Integer.valueOf(}iexp\texttt{);}\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \langle \pi\, v\ \texttt{=}\ \texttt{(Integer)}iexp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

Throughout this section all rules and approaches were given for the primitive type int. The boxing and unboxing conversions for the other supported numeric primitive types short, byte, and long result analogously. Note that for each type the cache function has to be different.

The assignment and casting rules for the primitive type boolean are correspondingly transferable. The accurate boxing rule looks a little different as boolean values cannot be used as indices to arrays:

$$\text{booleanBoxingAccurate}\ \frac{\begin{array}{l}\Gamma, c \doteq 0 \lor c \doteq 1, \mathcal{U}\neg(obj \doteq \text{null})\\ \mathcal{U}((bexp \rightarrow obj \doteq \text{TrueBooleanObject})\\ \qquad \land(\neg bexp \rightarrow obj \doteq \text{FalseBooleanObject}))\\ \vdash\\ \mathcal{U}\{obj\texttt{.value} := bexp \parallel obj\texttt{.<created>} := \text{TRUE} \parallel\\ \quad count_{\texttt{Boolean}} := count_{\texttt{Boolean}} + c\}\langle \pi\ \texttt{v=obj;}\ \omega\rangle\varphi, \Delta\end{array}}{\Gamma \vdash \mathcal{U}\langle \pi\, v\ \texttt{=}\ \texttt{(Boolean)}bexp\texttt{;}\ \omega\rangle\varphi, \Delta}$$

Herein TrueBooleanObject and FalseBooleanObject are rigid constants with the reference type Boolean.

5.3 Implementation Issues

The rules from section 5.2.2 that are responsible for making the boxing and unboxing context explicit cannot be realised as straight forward as it seems. For performance

reasons, KeY assumes every Java program to be correct w.r.t. the language specification, i.e. that a compiler accepts it. Many type checks can thus be omitted with reference to the language specification. Many rules do not check whether expressions are of the proper type as in Java prior to version 5, there was no reason to do so.

This changes when autoboxing and auto-unboxing are introduced to the language. Take for example the following rule that is responsible for the symbolic execution of assignment statements by transformation into an update. Here *loc* is a location and *se* is a simple expression.

$$\text{assignment}\ \frac{\Gamma \vdash \mathcal{U}\{loc := se\}\langle \pi\ \omega\rangle\varphi, \Delta}{\Gamma \vdash \mathcal{U}\langle \pi\ \texttt{loc = se;}\ \omega\rangle\varphi}$$

This standard rule may not be applied if, for instance, *loc* is of type int and *se* is of type Integer, in which case a cast operator is to be inserted. If the rule is applied anyhow, the situation pointed out in section 5.2.1 will emerge.

Built-in rules such as assignment have to be revised and conditions on the rule schema variables (such as *loc* and *se* in the case above) must be attached. Here, it must be ensured that either *loc* and *se* are both of primitive types or both of reference types.

The disadvantage of this approach is that quite a lot of the existing rules would have to be revisited and furnished with constraining conditions on the schema variables. This would "pollute" the existing core rules with details about the rather marginal topic of boxing.

The problem can also be addressed by giving the boxing context detection rules from section 5.2.2 a higher priority and make sure that they are applied – if applicable – before other rules are used, even may be used.

A third alternative is to "delegate" the process of identification of boxing contexts to an instance outside the calculus. All boxing and unboxing contexts are statically traceable and it is possible to mandate the semantic parser which builds the syntax tree to identify and mark boxing contexts.

5.4 Conclusion

Autoboxing and -unboxing is one of the topics of Java 5 that is vehemently discussed. While some see in it the right substitute for generic classes over primitive types, others fear the intransparency implied by its implicit application at various places throughout the code. Unforeseen `NullPointerException`s can arise at runtime if the null case is not always taken into consideration. Formal verification tools, in particular the KeY system, can be used to identify autoboxing and auto-unboxing situations and to ensure that no unwanted state appears that would lead to unexpected results.

For the actual boxing and unboxing processes, the creation of the appropriate value or object can be performed very accurately and in accordance with the language specification, no assumptions about the library implementation in use must be made.

Yet, the detection of the contexts in which this process has to take place is difficult and would involve refactoring at a lot of rules in the existing system.

6. Generic Classes

Genericity or parametric polymorphism is a fundamental paradigm in programming languages. But up to now, the Java programming language has not permitted to make generic declarations. This entailed that Java programs contained comparatively many explicit type conversions which brought along an increased error potential.

Using generic declarations reduces the number of explicit casting operations significantly and, hence, provides more type safety at compile-time. It is therefore little astonishing that already in the early days of Java, generic extensions were suggested. In 1997, the first proposals were published ([MBL97], [AFM97]). The language designers acted on these suggestions and included generic features into Java 5.

This chapter will, after introducing the new language features, propose a way how the JavaDL calculus can be extended to deal with generic classes. Generics in Java 5 are designed to be downward compatible with existing code which involves that they have some unintuitive properties. It will be shown that an augmented KeY calculus can be used to overcome some of these deficiencies.

6.1 Parametric Polymorphism

Parametric polymorphism allows to write code dealing with objects of different types in the same manner. The behaviour of generic code is independent of the datatype that is actually used. A list of items, for instance, behaves equally regardless of the type of the contained items. Thus, the behaviour of "a list of objects of type T" can be described generically without specifying the type T any further.

The parametrisation of generic entities can be either explicit or implicit. If it is explicit, every reference to a generic unit must be accompanied by a parametrisation of types. A language supports implicit parametric polymorphism if these type references can be omitted. In this case, the type parameters are statically computed from the expression by means of typing rules. This process is called *type inference*. In strongly typed functional programming languages such as Standard ML or Haskall, implicit parametric polymorphism is one of the main paradigms. Java supports implicit generic polymorphism and type inference only for generic methods.

Parametric polymorphism has been extensively examined in type theory. The typed lambda calculus often used to describe type systems in functional (but also imperative and object oriented) languages has been extended to a system that allows quantified types. This polymorphic typed lambda calculus, which is called *System F*, has been refined in 1985 by Cardelli and Wegner ([CW85], [Car97]) to permit bounds on type parameters, resulting in a calculus referred to as *System* $F_{<:}$.

System F adds a second order feature to the calculus as type variables are introduced and can be subject to λ-abstractions and quantifiers. The notion of universal types $\forall T$ and existential types $\exists T$ will be picked up and adapted to the context of the dynamic logic in section 6.3.6.

6.1.1 The Generic Decision in Java

Genericity is a common concept amongst modern statically typed languages. Hence, it is little astonishing that similar facilities were soon called for in Java, too.

There were basically three design paths the language designers could have followed. Every alternative had got its advantages and disadvantages.

Static code duplication produces a new copy of code for every instantiation of a generic class. It has the appropriate types instantiated for the type variables. Templates in C++ and the "NextGen" approach to generics in Java (see below) belong to this category.

Advantages: Efficient implementation, type safety, finite type system
Disadvantages: All instantiations have to be known at compile-time, inflexible, lots of code duplication.

Type safe dynamic generics create one class code only and store typing information so that type checks on the parameters can be performed at run-time. This is how parametric polymorphism is realised in Eiffel and C#.

Advantages: type safe extension, dynamic instantiation is possible
Disadvantages: little downward compatibility, the virtual machine would have to undergo rather drastic changes.

Dynamic generics with type erasure support generics only as a syntactic method for compile-time type checks and do not store this information for run-time. The structures are mapped to traditional Java discarding all information about type parameters.

Advantages: dynamic instantiation is possible, downward compatibility, virtual machine (basically) unchanged
Disadvantages: **Type safety is at stake**, inefficient extra type checks

PolyJ (proposed in [MBL97]) was a first approach to the subject, Pizza ([OW97]) is a framework that extends the Java language by generics amongst other features. The original proposal left open, whether dynamic or static generics should be used for implementation. In 1998 Bracha and Odersky proposed Generic Java (GJ) in [BOSW98] which refined Pizza's generic ideas and was mainly designed to be compatible with previous Java versions. GJ introduced the idea of dynamic generics

with type erasure in order to change the specification of the Java virtual machine as little as possible.

The Java Community Process JCP which is responsible for the development of the language issued the Java Specification Request JSR14 in which they followed the ideas of Bracha and Odersky and point three of the enumeration above.

The process of removing typing information at compile-time is known as **type erasure**, a well-known aspect in type theory. As soon as type checking for a statically type checked language has finished without errors, typing information may be discarded; it is no longer needed, as the program has proven to be typesafe at compile-time already.

To understand the problem type erasure imposes on the language, something must be said about type safety. Typesafe programming languages can be divided into two general categories:

- Statically checked languages check the type consistency at compile-time. Any program is guaranteed to run without error, the language is statically safe. Purely statically typed languages include strongly typed functional languages like SML and Haskell.
- In dynamically checked languages, locations do not have fixed static types but can hold any value of a variety of types. To ensure that all operations are permitted, typing information must be present at run-time to allow type checks. Most scripting languages are purely dynamically typed.

While Java generally is considered to be a statically checked language, it should better be seen as a hybrid language. While most expressions are statically type-checked by the compiler, there are constructs that need dynamic typing. These are for instance the explicit cast operator or the instanceof operator. The compiler provides this information for them.

In the context of generic classes, the type parameters are a part of the dynamic type information about an object. In order to check a cast operation like `(List<Integer>) listObject` at run-time, the object `listObject` needs to store information about its type and about the parameters of the generic type.

Unfortunately, because of the design decision, all information about type parameters to generic classes are discarded when compiling. This has several implications on the language. First, certain statements that can be formulated intuitively like instanceof operations on parametrised types are not possible and lead to compile-time errors. Second and more severe is the fact that certain cast operations which should lead to an exception at run-time are executed without complaint and lead to an unsound state which can give raise to an unexpected exception at some later point in time.

Hence, while generics improve type safety immensely in a *static* context, they change it for the worse in a *dynamic* context.

An approach that evolved at around the same time as GJ proposed a system which does not suffer from this deficiency and keeps nonetheless backward compatibility

with traditional Java. "NextGen" ([CG98], [SC06]) uses a special class loader to generate parametrised types at run-time. It runs on any virtual machine and overbears many limitations that generics have in Java 5.

The Java community has not yet settled upon the unfortunate decision. There are signs that indicate that the generic decision has not been made once and for all. For instance, the language specification [GJSB05] justifies the design decision in §4.7 with:

> The price of migration compatibility is that a full and sound reification[1] of the generic type system is not possible, at least while the migration is taking place.

This last restriction slightly suggests that in the future the generic type system might be changed so that run-time access to generic types will be available. Neal Gafter, a co-designer of Java 5 at SUN, has recently published a note [Gaf06] in which he writes:

> Many people are unsatisfied with the restrictions caused by the way generics are implemented in Java. Specifically, they are unhappy that generic type parameters are not reified: they are not available at runtime. [...] It isn't too late to add reified generics to Java. [...] In the ideal world, we wait until every bit of Java code in the world has been modified to use generics safely, and then go through a transition in which we start recording type parameters at runtime [...].

I think, there is sufficient evidence that in a future version of the language, the generic type system will be revised and the type information made available at run-time. Once a migration period is over, the drawbacks of the design decision could then be overcome.

6.1.2 Syntax and Semantics of Generics in Java

In comparison to the other enhancements of Java 5, generics have by far the greatest impact on the grammar. The syntactical changes add type arguments, type parameters and type variables to the language.

The syntax and semantics of generic classes will be introduced here only briefly. A very good introduction is provided by the leader of the development team Gilad Bracha in [Bra04]. Details on the semantics can be found in the specification [GJSB05] §8.

With generics, a class declaration has now the basic appearance

```
class ClassName<TypeParameters> extends ...  implements ...  {...}
```

with *TypeParameters* a comma-separated list of type parameters. A formal type parameter is an identifier (by convention a single capital letter is used) optionally

[1] Types that are completely available at run-time are known as reifiable types

followed by constraining bounds. Bounds are the keyword `extends` followed by one or more (possibly parametrised) reference types separated by `&`.

The type parameters that appear in the class declaration are the type variables that may be used within the class declaration in place of usual types in the class body or in 'implements' and 'extends' clauses. The following example class declaration makes use of the new features:

Java 5

```
1 public class Class1<T extends Class2 & Interf1, U> implements Interf2<T> {
2
3     U attribute;
4
5     T method(U e) { /*...*/ }
6
7 }
```

Java 5 – 6.1

The generic class `Class1` has two type parameters T and U. The parameter T is constrained so that when instantiating `Class1`, the formal parameter T must be replaced by a type that is a subtype to both class *Class2* and interface *Interf1*. The type parameter U is unconstrained, it can be instantiated by any type that is a subtype of `Object`.

Generic types can be thought of as "functions" of types to types; they have formal parameters that can then be instantiated with various types. So when using generic types, the formal type parameters are always concretely instantiated with reference types. The application of a generic type to types is called a **parametrised type** for obvious reasons.

Generic classes do not have covariant inheritance like arrays do. While $A \sqsubseteq B$ implies that $A[\,] \sqsubseteq B[\,]$, it does not imply that $G\langle A\rangle \sqsubseteq G\langle B\rangle$ for a generic class G. A phenomenon similar to the appearance of `ArrayStoreException`s when assigning values to an index in an array has thus been avoided for generics. To have a common supertype for $G\langle A\rangle$ and $G\langle B\rangle$ anyhow, the language designers included the generic wildcard instantiation $G\langle\mathbf{?}\rangle$ which is supertype to any concrete instantiation $G\langle C\rangle$.

Wildcard types can also be bounded using the `extends` or the `super` keyword. `extends` constrains the arguments to certain subtypes and `super` to supertypes of the mentioned type. The type hierarchy in Java 5 has grown quite complicated in comparison to traditional Java because of the co- and contravariant abstract types that have been added. Possibly bounded wildcards make the type system both flexible and complex.

The theory behind wildcards is rather elaborate and has been introduced in [THE+04] and [IV02]. The typed lambda calculus System $F_{<:}$ which allows bounded polymorphism has been used to formally describe bounded wildcards in Java. The soundness of the typing system that involves wildcards has been proved in [TEH05].

Figure 6.1 shows an extract of the type hierarchy for a scenario in which there is a generic class $G\langle\cdot\rangle$ with one type parameter and the non-generic classes $A, A2$ and B with $A \sqsubseteq A2$. Generics do not have the covariant subtyping property, but the according bounded wildcard types can be used to have typesafe access. All instances $G\langle$`? extends ...`$\rangle$ are abstract types of which no instances can be created.

Fig. 6.2 shows another part of the object-type hierarchy. While the wildcards using an 'extend'-clause have the covariant subtyping property

$$A \sqsubseteq B \implies G\langle \texttt{? extends } A\rangle \sqsubseteq G\langle \texttt{? extends } B\rangle,$$

wildcards using the 'super' keyword are contravariantly related as in

$$A \sqsubseteq B \implies G\langle \texttt{? super } B\rangle \sqsubseteq G\langle \texttt{? super } A\rangle.$$

All types that use the wildcard as a direct type parameter like $G\langle ?\rangle$ are abstract types and cannot be instantiated. If the wildcard is not a *direct* argument, for instance in $G\langle G\langle ?\rangle\rangle$, the type is dynamic and can be instantiated. It does e.g. make sense to create a list of lists of unknown type.

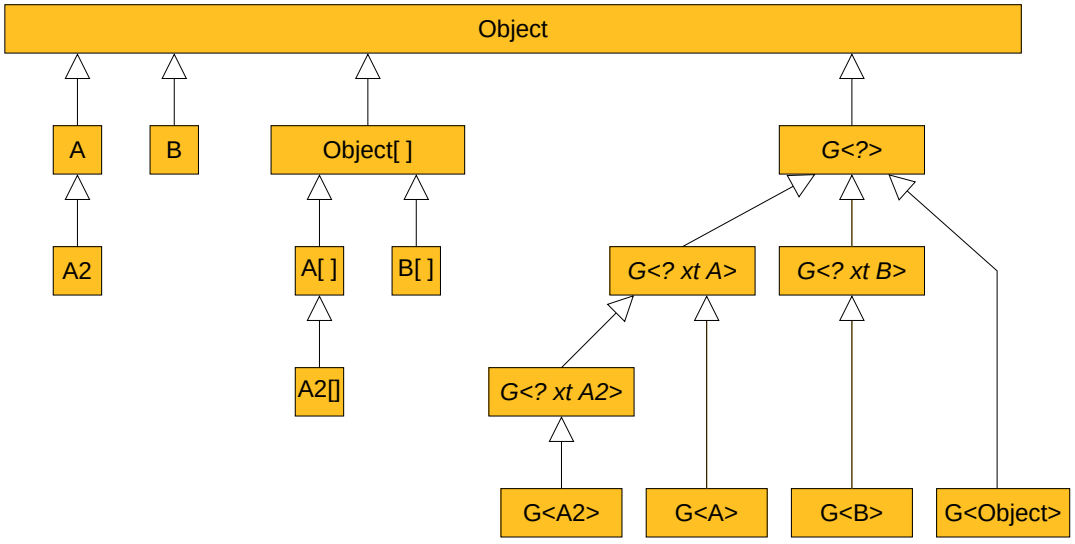

Figure 6.1: Covariance for generic types. Abstract types are italic. 'xt' is short for 'extends'.

The semantic of a generic class G can be described best as a schema of declaration. Every instantiation $G\langle A_1, \ldots, A_n\rangle$ of G with types A_i rather than wildcards behaves as if there was class `G<`A_1`, ..., `A_n`>` defined with the type parameters substituted by the actual parameters.

The types and signatures of members of types with wildcard parameters depend on the boundaries in use and involve a lot of covariant and contravariant relationships. Section 6.3.2 will deal with such a situation.

Methods or constructors can have generic parameters, too. The parameter is then to be declared in front of the return type. Let us look at an example from the Java Collection Framework in listing 6.2: The method `singletonList` is polymorph, it takes an argument of arbitrary type and return a list of the same type.

In the case of generic methods, it is not always necessary to state the type parameter explicitly. Under certain circumstances, the compiler is able to deduce the type for the parameters and can instantiate the parameters on its own. This is the only case in which Java 5 supports implicit parametric polymorphism by type inference. So

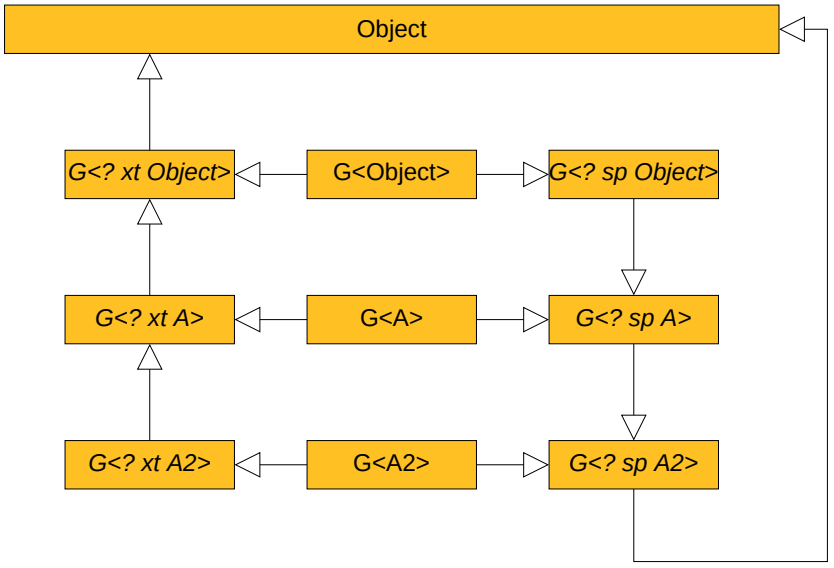

Figure 6.2: Covariant and contravariant supertypes. 'xt' is short for 'extends' and 'sp' is short for 'super'

the method stated above can be used either explicitly or implicitly typed. Both invocations are correct.

— Java 5 fragment

```
public static <T> List<T> singletonList(T o) {
    return new SingletonList<T>(o);
}
```

```
List<Integer> list1 = singletonList(new Integer(1));
List<Integer> list2 = <Integer>singletonList(new Integer(2));
```

Java 5 fragment – 6.2

6.1.3 Unchecked Operations

Consider that you plan to write a class `Cast` that casts any object type to a specific type and then allows access to it.

— Java 5

```
1  public class Cast<E> {
2
3      E item;
4
5      public Cast(Object o) {
6          item = (E) o;
7      }
8
9      public E get() {
10         return item;
11     }
12
13     public static void main(String[] args) {
14         Cast<Integer> cast = new Cast<Integer>("String");
```

```
15          System.out.println(cast.get().intValue());
16      }
17
18  }
```

Java 5 – 6.3

This program does compile, but the compiler issues the warning

```
Cast.java:6: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: E
```

which is an *unchecked conversion warning*. When running this program we expect a `ClassCastException` in line 6 as the type variable E is instantiated to `Integer` and o is of type `String` when the constructor is called in line 14. But instead, we obtain the `ClassCastException` in line 15 though this line does not contain a statement that may issue a `ClassCastException` in traditional java.

The pragmatic reason for this is as follows: For the sake of downward compatibility the compiler throws away all information about parametrised types, performs type erasure, and uses the raw types. This way no cast that involves type variables can be checked at run-time.

The effect that arises here is called "heap pollution"[2] (JLS 4.12.2.1):

> It is possible that a variable of a parametrised type refers to an object that is not of that parametrised type. This situation is known as heap pollution. This situation can only occur if the program performed some operation that would give rise to an unchecked warning at compile-time.

In fact, every operation that contains explicit (or sometimes even implicit) casts to a parametrised type or a type that contains type variables is an unchecked operation. Theoretically it is possible that such a statement can pollute the heap. If there are no unchecked conversions, type safety is ensured.

Even if the heap becomes polluted at some point, the type system will never be really annulled as an object in memory that has a type will never be treated as if it had an incompatible type. This could for instance happen within the C++ programming language since it is not typesafe. Instead exceptions will be thrown at locations which would never do so in a completely typesafe environment. These exceptions are sometimes called "untrapped" or "deferred" errors in literature.

The Java Collection Framework[3] contains 253 unchecked warnings which mainly arise from handling generic arrays and from downward compatibility issues involving raw types.

Section 6.1.3 deals with handling such cases in the calculus.

[2] I will refer to heap pollution also if a variable of a type variable type refers to an object that is not compatible in its context.

[3] more precisely all classes in the package `java.util`

6.2 Type Erasure in the Calculus

When programs that use generics have to be dealt with in the context of formal verification, there are two general possibilities to proceed:

1. Transform the program so that the result does not make use of generics. This transformation ought to be done as naturally as possible. The default calculus can be used to verify the converted programs.
2. Change the syntax and semantics of the calculus so that it can handle generic classes as well.

The specification of the Java Virtual Machine in release 5 does not differ very much from its predecessor. In fact, no structural additions were made to cover the generic extension.

This is why there is a canonical code transformation that transforms source code which employs generic classes to code that does not make use of genericity. But this transformation cannot be done within the KeY-system itself but has to be performed prior to presenting the code to the calculus and after parsing the source code. The calculus itself remains without change, which of course disallows to use generics explicitly in modalities.

The source code transformation is composed of two steps:

- First insert explicit type casts in places where generic type variables are involved.
- Then remove all traces of generic class and method definitions and type variables. Replace all parametrised types by the raw type (i.e. perform type erasure).

For any generic class the following search must be performed:

Let G be a generic class with (possibly bounded) type variables. Locate all occurrences for a method $G.m(\dots)$ that has a return type which is either a type variable or an array over a type variable. Surround every such application with an explicit cast to the static type of the expression.

At first glance it does not make sense to explicitly cast an expression to the type it already has. But when later type variables are replaced with more general types during type erasure, it is hereby ensured that the result is still of the same type as before.

In a similar manner, explicit casts have to be inserted wherever there is a reading access to an attribute $G.a$ of class G and the type of this attribute is a type variable or an array over a type variable.

The transformation is only briefly described here and becomes significantly more difficult under certain circumstances, for instance if multiple boundaries are given on the static type of the expression upon which m is invoked. It is however always statically computable as a compiler must be able to calculate the results also.

Let us have a look at an example that clarifies this process. Class C is a generic class with an attribute *attr* and a method *meth*() of which the types depend on the type variable T.

Java 5

```
1  class C<T> {
2      T attr;
3      T meth() { return attr; }
4
5      void n() {
6          C<Integer> ci = new C<Integer>();
7          Integer a = ci.attr;
8          Integer l = ci.meth();
9
10         C<?> co = ci;
11         Object o = co.a;
12     }
13 }
```

Java 5 – 6.4

Performing the introduction of the explicit casts affects lines 7, 10, and 11, the result is as follows.

Java 5

```
1  class C<T> {
2      T attr;
3      T meth() { return attr; }
4
5      void n() {
6          C<Integer> ci = new C<Integer>();
7          Integer a = (Integer) ci.attr;
8          Integer l = (Integer) ci.meth();
9
10         C<?> co = ci;
11         Object o = co.attr;  // no cast needed − types compatible
12     }
13 }
```

Java 5 – 6.5

The second step, the type erasure, is described rather easily: Change every class declaration $G[T_1, \ldots, T_n]$ with n possibly bounded type variables into a declaration G without type variables. Replace every appearance of a type variable by its boundary (or `java.lang.Object` if none given). Reduce every parametrised type $G\langle C1, ..., C_n\rangle$ to the raw type $|G|$ regardless of the parameters and whether wildcards are used or not.

The following program is equivalent to the generic program given in listing 6.4 w.r.t the language specification. Now it is clear, why the explicit cast operations had to be inserted: Without them the program would not compile due to typing errors.

Java 2

```
1  class C {
2      Object a;
3      Object m() { return attr; }
```

```
4
5       void n() {
6           C ci = new C();
7           Integer a = (Integer)ci.attr;
8           Integer l = (Integer)ci.meth();
9
10          C co = ci;
11          Object o = co.attr;
12      }
13  }
```

JAVA 2 – 6.6

This transformation also explains the deferred error messages from section 6.1.3 which seemed to appear at arbitrary locations. They showed up at places in which additional casts had been inserted and had failed at run-time.

6.3 Extending the JavaDL Calculus to Generic Types

The approach that has been pointed out in the previous section cannot – albeit being complete and correct – be satisfactory. It merely expresses the new structures by old means, and the new expressivity is lost by the transformation. But generic classes are a new concept in the Java language, and it is desirable to transfer the novelty to the calculus also. The JavaDL calculus will need to meet some major changes to be able to handle all eventualities with generic classes.

Thus, to make things not too complex, restrictions are set on the application of the generic extension within this section: Generic classes are limited to use only unbounded type variables, and wildcards need to be without boundaries, too. Boundaries to type wildcards are a non-trivial matter from the type system's point of view, they even lead to the undecidability of some questions in type theory and, hence, should be the matter of further studies.

I will also disallow to use raw types, i.e. is generic classes in their unparametrised appearance. Raw types may disappear in future versions, so disregarding them is no big loss (cf. [GJSB05] §4.8). Generic methods will also not be addressed by this approach.

For the sake of better readability let $G\langle\mathbf{A}\rangle := G\langle A_1, \ldots, A_n\rangle$ and similar be abbreviations for parametrised types of a generic class G with $n \geq 1$ type arguments.

6.3.1 A type System with Generic Classes

In this section the existing framework of the type system as it has been pointed out in section 2.2.1.1 will be enriched, so that it also comprises generic types.

The type system in traditional Java prior to release 5 already had the potential to contain an infinite number of types albeit only finite many classes could be declared. A program can make use of arrays of arbitrary depth n so that

$$\texttt{int}\underbrace{\texttt{[][]...[]}}_{n \text{ times}}$$

is a perfectly valid type in Java for any $n \geq 0$. However, there was no way – if not using reflection methods, which are explicitly not supported in KeY – to use types that are not literally mentioned in the program source code.

With the introduction of type variables this changes. The following program makes use of arbitrarily many types which are not mentioned in the source code.

Java 5

```
1  public class Inf<E> {
2
3      Inf() { }
4
5      Inf<Inf<E>> step() {
6          return new Inf<Inf<E>>();
7      }
8
9      public static void main(String[] args) throws Exception {
10
11     Inf<?> infinite = new Inf<Object>();
12
13         while(System.in.available() == 0) {
14             infinite = infinte.step();
15         }
16     }
17 }
```

Java 5 – 6.7

Though the set of literally mentioned static types only contains `Inf<Object>` and the wildcard type `Inf<?>` the set of dynamic types appearing at run-time can include the type

$$\texttt{Inf<}\underbrace{\texttt{Inf<Inf<...<Inf<}}_{n \text{ times}}\texttt{Object>>...>>>}$$

for any $n \geq 0$. The program terminates as soon as the user presses the enter key, so the maximum depth n of the generic parametrisation cannot be computed in advance (assuming unlimited memory space).

This situation, which is called *polymorphic recursion*, occurs as generic classes can be instantiated with a type variable as type argument.

The very same phenomenon can also be observed for arrays. Listing 6.8 shows a program generating array types of arbitrary depth. Note however that it is not possible to actually instantiate an array over a type variable, so only the null reference can be used here. Section 6.6 discusses the topic of generic arrays extensively.

Java

```
1  public class InfArray<E> {
2
3      InfArray() { }
4
5      Inf<E[]> step() {
6          return new Inf<E[]>();
7      }
8
9      E[] getArray() { return null; }
```

```
10
11      public static void main(String[] args) throws Exception {
12          InfArray<?> infinite = new InfArray<Object>;
13          while(System.in.available() == 0) {
14              infinite = infinte.step();
15          }
16          System.out.println(infinite.getArray());
17      }
18
19  }
```

JAVA – 6.8

Obviously, a program can make use of an unbounded number of types at run-time. This implies that the set of types which are available in the model must either be an infinite set, or it must be a set which grows in size as the verification process goes on.

6.3.1.1 An Infinite Type System

A JavaDL program P with generics is a JavaDL program (cf. section 2.2.1.5) with the following additional properties:

- P may make use of the generic extensions of the Java language as defined in the JLS. The other novelties (boxing, enum, etc.) may not be used.
- Any type variable within the declaration of a generic class is unconstrained.
- Any wildcard '**?**' is unconstrained.
- No generic methods are defined.

Only some types can be used as parameters to generic classes. In order to define the compatibility of a type system to a program P with generics, the subset of types that can be used as type parameters has to be defined.

Definition 6.1 (Reference types) Let $(\mathcal{T}, \sqsubseteq)$ be a type system which contains the distinct types `Object` and `Null`. Then the set $\mathcal{J}$ of reference types is defined by

$$\mathcal{J} := \{T \in \mathcal{T} : T \sqsubseteq \texttt{Object} \wedge T \not\sqsubseteq \texttt{Null}\}.$$

The set of reference types $\mathcal{J}$ represents the set of types that can be spoken of in Java and that are not primitive types. It includes arrays over primitive or reference types, non-generic classes, any parametrised instance of any generic class, but not the raw generic classes themselves.

In addition to $\mathcal{J}$ the type system contains some other types: the types $\top$ and $\bot$, types that represent the primitive datatypes, a Null-type, etc. The following definition does not deal with that part of the type set and assumes it to be reasonable. The set $\mathcal{J}$ of reference types can be adjusted exactly to the needs of a program P.

Definition 6.2 A type system $(\mathcal{T}, \sqsubseteq)$ is compatible with a generic JavaDL program P if the set of reference types $\mathcal{J} \subset \mathcal{T}$ is the smallest set which fulfils:

- $C \in \mathcal{J}$ for any non-generic class C,
- for any generic class G with $n \geq 1$ type variables every instance $G\langle A_1, \ldots, A_n\rangle \in \mathcal{J}$ with $A_i \in \mathcal{J} \cup \{\texttt{?}\}$ is a reference type,
- For any reference or primitive type $T \in \mathcal{J} \cup \{\texttt{int}, \texttt{boolean}, \ldots\}$ and any dimension $d \geq 1$ the array type $T[]^d \in \mathcal{J}$ is also a reference type, and
- for any two types $A, B \in \mathcal{J}$ the relation $A \sqsubseteq B$ holds iff A is a subtype of B in the sense of the Java language after JLS §8.1.5

Even if there is no generic class present, the type system still needs to be infinite with $\texttt{Object[]}^d \in \mathcal{J}$ a reference type for any d. For every generic class every possible instantiation is present in the type hierarchy. This is a conservative proceeding as not every generic class allows type recursion. But a costly static analysis would be needed to determine which types can be infinitely nested and which cannot.

6.3.1.2 A Growing Type System

Instead of using an infinite type system that a priori covers all eventually upturning types, one might prefer a type system that initially contains only the literally mentioned static types and dynamically adds those new types that appear at run-time.

This section will show that this approach is **not** sound as it violates not only the rule of a constant type set but along with that also the paradigm of a constant domain as for every new type new objects are injected into the domain.

To proof the unsoundness, we take a rule that needs a constant and finite type hierarchy to be correct. The rule schema expandDynamicType is such a rule. It introduces a case distinction on the dynamic type of an arbitrary term based on its static type. Let $t \in \mathrm{Trm}_T$ be a term of a reference type $T \sqsubseteq \texttt{Object}$ and $S_1, \ldots, S_n \in \mathcal{T}_d$ with $S_i \sqsubseteq T$ an enumeration of all dynamic subtypes of T, then the application of the rule

$$\text{expandDynamicType}\ \frac{\Gamma, t \in_1 S_1 \vee t \in_1 S_2 \vee \ldots \vee t \in_1 S_n \vee t \doteq \text{null} \vdash \Delta}{\Gamma \vdash \Delta}$$

adds a case distinction for the dynamic type of t based on its static type T to the antecedent of the current sequent. The term t must be either a direct instance of one of the dynamic subtypes of T or hold the null reference.

Now assume that the class stated in listing 6.7 is given. The type system $\mathcal{T}_0$ is the initial type system and contains all types which appear verbally in the program: $\mathcal{T}_0 = \{\texttt{Object}, \texttt{Inf<?>}, \texttt{Inf<Object>}\}$. Let c:`Inf<?>` be a program variable and η the Java code

$$\eta = \text{“}\texttt{c = new Inf<Object>().step();}\text{”}$$

which assigns one iteration step to c. A new dynamic type `Inf<Inf<Object>>` is introduced when η is symbolically executed. The variable c will then hold a reference to an object of this type even though it has never been literally mentioned at all. The initial type system $\mathcal{T}_0$ has been silently augmented by one new type to become the set $\mathcal{T}_1 = \mathcal{T}_0 \cup \{\texttt{Inf<Inf<Object>>}\}$ after the execution of η.

The result of the rule expandDynamicType depends on the underlying type set. In case of the initial type system $\mathcal{T}_0$ the type distinction for c:`Inf<?>` reads

$$A(c) := c \doteq \text{null} \vee c \sqsubseteq_1 \texttt{Inf<Object>}.$$

Within $\mathcal{T}_0$ every object of type `Inf<?>` must either be an object of type `Inf<Object>` or be the null reference. The answer to the question whether $A(c)$ is valid depends on the type system. It is valid in any Kripke structure with type set $\mathcal{T}_0$ but not valid (yet satisfiable) in Kripke structures with $\mathcal{T}_1$ as underlying type set.

This is a serious problem as the time at which the rule that augments the type system is performed cannot be fixed. Figure 6.3 shows an example of a situation in which the order of the applied rules decides whether a formula can be proven valid or not. In 6.3a the formula $A(c)$ is proven to be valid. This is – as we have seen – correct if the type system $\mathcal{T}_0$ is assumed. If, however, prior to the type distinction the program η is evaluated (like in 6.3b in the formula $[\eta]\,\text{false}$), the type system "switches" to $\mathcal{T}_1$ and $A(c)$ can no longer be proven because one more dynamic type is present.

$$\dfrac{\dfrac{}{(\text{close})\quad A(c) \vdash A(c), [\eta]\,\text{false}}}{(\text{extDynType})\quad \vdash \boxed{A(c)}, [\eta]\,\text{false}}$$

(a) If $A(c)$ is evaluated first

$$\dfrac{\dfrac{A(c) \vee c \sqsubseteq_1 \texttt{Inf<Inf<Object>>} \vdash A(c), \text{false}}{(\text{extDynType})\qquad \vdash A(c), \text{false}}}{(\ldots)\qquad \vdash A(c), \boxed{[\eta]\,\text{false}}}$$

(b) If $[\eta]\,\text{false}$ is evaluated first

Figure 6.3: With a growing type system, the order of evaluation decides whether a formula can be proven or not.

A growing type system provides a finite type system at any time, but it violates the constant domain paradigm (observation 2.16) which has been set in the KeY environment and which has to be held up despite the introduction of generic types. In the following the type system will therefore be the infinite type system presented in section 6.3.1.1.

6.3.2 Member Fields of Wildcard Types

Consider the following declaration of a generic class G.

— Java 5 —

```
1 class G<A> {
2
3     A x;      // a field whose type is a type parameter.
4
5     static void m() {
6         G<Integer> g = new G<Integer>();
7         G<?> h;
8         h = g;
9         g.x = new Integer(0);
```

```
10          System.out.println(g.x);
11          System.out.println(h.x);
12      }
13  }
```

JAVA 5 – 6.9

In the method `m` two objects g and h are declared. While the type of g is a wildcard-free instantiation of the generic class G, the type of h contains a wildcard. Due to the assignment in line 8, both variables point to the same object.

What is the static type of the expressions $g.x$ and $h.x$ in lines 10 and 11? While the instantiation of G with the type `Integer` makes clear that $g.x$ is an Integer, the instantiation $G\langle\texttt{?}\rangle$ for h only implies that there is a type T so that $h.x$ is of type T. As long as $g = h$ holds, both references $g.x$ and $h.x$ always refer to the same reference though they are of different static type.

Wildcard types in Java 5 have an interesting feature which does not occur elsewhere in the language: Covariant subtyping of member fields. In traditional Java the definition of a class attribute in both a superclass and a subclass results in the latter declaration hiding the former. In case of generic classes, it may be that a supertype has an attribute of a more general type than the subtype, here $G\langle\texttt{Integer}\rangle \sqsubseteq G\langle\texttt{?}\rangle \implies \text{typeof}(G\langle\texttt{Integer}\rangle::x) \sqsubseteq \text{typeof}(G\langle\texttt{?}\rangle::x)$.

The question whether the formula

$$\vdash \langle\texttt{G.m();}\rangle(h.x \doteq g.x) \tag{6.1}$$

holds, unveils a problem: the terms $h.x$ and $g.x$ do not refer to the same function in the logic. Hence, for the purpose of this section, we have to occasionally drop the convenient notion $o.a$ for the access to a member field a of the reference o of type C. Instead the original formulation as a function application $C::a(o)$ will be used, so that formula (6.1) reads

$$\vdash \langle\texttt{G.m();}\rangle \Big(G\langle\texttt{?}\rangle::x(h) \doteq G\langle\texttt{Integer}\rangle::x(g)\Big) \tag{6.2}$$

Since g and h are equal as references, both references $h.x$ and $g.x$ are the same as well, yet the same location $g.x \leftrightarrow h.x$ can be accessed once as an `Integer` and once as an `Object` location in a covariant manner. But in the calculus it is not deducible that the relationship

$$G\langle\texttt{?}\rangle::x(g) \doteq G\langle\texttt{Integer}\rangle::x(g) \tag{6.3}$$

must hold. This leads to the introduction of a new rule fieldProxy in (6.4) later in this section that covers the issue that a field reference may be used in place of another one. The rule has to recognise accesses to the same member field via different types and replace them by one way to access them.

In the preceding example where the attribute x was addressed once by $G\langle\texttt{Integer}\rangle::x$ and once by $G\langle\texttt{?}\rangle::x$, the application of fieldProxy for the term $g.x$ yields

$$\text{(fieldProxy)}\ \frac{\text{(close)}\ \dfrac{}{\vdash G\langle\texttt{Integer}\rangle::x(g) \doteq G\langle\texttt{Integer}\rangle::x(g)}}{\vdash G\langle\texttt{?}\rangle::x(g) \doteq G\langle\texttt{Integer}\rangle::x(g)}$$

so that the relationship can be proved because the access function $G\langle\textbf{?}\rangle::x$ has been replaced by $G\langle\texttt{Integer}\rangle::x$.

In the programming language the covariant subtyping may only be performed since the field $h.x$ cannot be written to. Being of a more general type, an incompatible reference might be stored in $h.x$ (and therefore in $g.x$) which would destroy type safety. So attempting to write to a field which has a wildcard type leads to a compile-time error.

If in the logic one was allowed to assign a value to $G\langle\textbf{?}\rangle::x(g)$ within an update, the following sequent could be proven to be valid even though the formula without the negation (†) can be proven using the rule fieldProxy. There would be an inconsistency.

$$\begin{array}{ll} & o_1 \sqsubseteq_1 \texttt{Object}, o_2 \sqsubseteq_1 \texttt{Object}, o_1 \neq o_2 \\ \vdash & \\ & \{G\langle\textbf{?}\rangle::x(g) := o_1 \parallel G\langle\texttt{Integer}\rangle::x(g) := o_2\} \\ & \quad \neg^{(\dagger)}(G\langle\texttt{Integer}\rangle::x(g) \doteq G\langle\textbf{?}\rangle::x(g)) \end{array}$$

This problem can be resolved by making the field access function $G\langle\textbf{?}\rangle::x : G\langle\textbf{?}\rangle \to \texttt{Object}$ in the logic a read-only function that may not be used as location in an update. The function $G\langle\texttt{Integer}\rangle::x : G\langle\texttt{Integer}\rangle \to \texttt{Integer}$, however, needs to remain writable. Setting a function read-only is a mere syntactical limitation on the function symbol, it does not need to be rigid. The term $G\langle\textbf{?}\rangle::x(g)$ always refers to the same reference as $G\langle\texttt{Integer}\rangle::x(g)$ which is not a rigid function so that $G\langle\textbf{?}\rangle::x(g)$ can also change its content with states.

With the general idea about the shared attributes sorted out, it remains to be defined which field access functions have to be available for which types in the type hierarchy. The auxiliary notion of the largest instance of an attribute is needed to describe the types in which attributes are defined read-only or writable.

Definition 6.3 (Largest instance of an attribute) Let G be a generic class and $G.attr$ an attribute in G. Then the parametrised type $G\langle\mathbf{A}\rangle$ is a largest instance for $attr$ if and only if for all parametrised types $G\langle\mathbf{B}\rangle \sqsubseteq G\langle\mathbf{A}\rangle$ the static type of $G\langle\mathbf{B}\rangle.attr$ is the same as $G\langle\mathbf{A}\rangle.attr$.

The set of the largest instances for $G.attr$ is denoted as $\Theta(G.attr)$.

A parametrisation is a largest instance if all instantiations that are more specific (have less wildcards) have the same type for the attribute $attr$. This definition is illustrated by an example with a generic class X that has three attributes a_0, a_1 and a_2:

Java 5

```
1 class X<A,B> {
2     int a0;
3     G<A,A> a1;
4     G<B,A> a2;
5 }
```

The largest instances for the attributes differ. The attribute a_0 is accessible with integer type in all instances of X so that $\Theta(X.a_0) = \{X\langle ?, ?\rangle\}$. For a_1 the type depends on the first type variable, so the largest types need to have a fixed first argument: $\Theta(X.a_1) = \{X\langle T, ?\rangle : T \in \mathcal{J}\}$. The type of the attribute a_2 depends on both type variables, so that the set of most generic instances is the set of all instances without wildcards $\Theta(X.a_2) = \{X\langle T, U\rangle : T, U \in \mathcal{J}\}$.

Definition 2.9 in chapter 2 defines which logic functions must be present in the signature for a non-generic JavaDL program. This definition can now be adapted for programs that contain generic classes.

Definition 6.4 (Compatible signatures)
A JavaDL signature (VSym, FSym, PSym) is **compatible** with a JavaDL program P with generics if:

1. for each static field declaration "$Ty\ id$" within a (generic or non-generic) class C there is a function

   $$C{::}id{:}Ty \in \text{FSym}$$

   in the set of function symbols,

2. for each non-static field declaration "$Ty\ id$" within a non-generic class C there is a non-rigid function

   $$C{::}id{:}C \to Ty \in \text{FSym}$$

   in the set of function symbols, and

3. for each non-static field declaration "$Ty\ id$" within a generic class G and $G\langle\mathbf{A}\rangle$ a supertype of a largest instance T of $G.id$, i.e. $T \sqsubseteq G\langle\mathbf{A}\rangle$, $T \in \Theta(G.id)$, there is a non-rigid function

   $$G\langle\mathbf{A}\rangle{::}id{:}C \to Ty' \in \text{FSym}$$

   in the set of function symbols. The function is a read-only symbol that may not be used as left hand side in updates if $T \notin \Theta(G.id)$ is not a largest instance, but a proper supertype of a largest instance.

The type Ty' is the result of the instantiation of the type variables $V_1, \ldots, V_n$ of Ty with the parametrisation $A_1, \ldots, A_n$ of $\mathbf{A}$:

$$Ty' = \begin{cases} \texttt{Object} & \text{if } \{V_i \leftarrow A_i\}Ty = ? \\ \texttt{Object[]}^d & \text{if } \{V_i \leftarrow A_i\}Ty = ?[]^d \text{ for a } d \geq 1 \\ \{V_i \leftarrow A_i\}Ty & \text{otherwise} \end{cases}$$

If the type Ty of id in the last case is a type parameter, the function that has to be introduced would have to have a type variable type which is obviously not correct. To prevent this, the type Ty' is used.

The rule fieldProxy still remains to be defined. The idea is to replace all accesses to fields via various access functions by the access function that is as specific as possible.

If an access function of a parametrised type is used on a more specific argument (whose type has less wildcards), this access can be replaced. Let *attr* be a non-static attribute of class generic G, and the attribute access term $G\langle\mathbf{B}\rangle::attr(x)$ appear in some formula in $\Gamma \cup \Delta$. The argument $x \in \mathrm{Trm}_{G\langle\mathbf{A}\rangle}$ of this term is of a more specific type $G\langle\mathbf{A}\rangle \not\sqsubseteq G\langle\mathbf{B}\rangle$ than the parameter of the access function. In this case, the access $G\langle\mathbf{B}\rangle::attr(x)$ acts as a proxy for the actual attribute access $G\langle\mathbf{A}\rangle::attr(x)$ and may therefore be replaced.

$$\mathsf{fieldProxy}\ \frac{G\langle\mathbf{A}\rangle::attr(x)}{G\langle\mathbf{B}\rangle::attr(x)} \tag{6.4}$$

While it is still necessary to keep the different access functions because of their different types, this rule puts them into relation and makes it possible to use them simultaneously.

This chapter shows why it is not sufficient any more to have unique identifiers in the Java program. There would be a name clash as for one identifier declaration there may be many corresponding function symbols in the logic. The function symbols need to carry the class they belong to in their names.

The entering of generic classes brings along the introduction of covariant class attributes. This concept can be covered in JavaDL by establishing a rule fieldProxy that relates covariant fields and by field access functions that are read-only, i.e. that cannot be subject to a value change by an update.

6.3.3 Type Metatypes

The polymorphic lambda calculus (cf. section 6.1) supports existentially quantified types to express that a term has some type that is not precisely given. A list *intlst* of integers can therefore be considered in such a calculus as a list of some type α and be denoted as

$$intlst : \exists\alpha.\alpha \text{ list},$$

thus, "there is a type α so that *intlst* has type list of α."

If we leave the lambda calculus and try to transfer this notion to JavaDL, the same situation with *intlst* a `List<Integer>` object would produce the following formula:

$$\exists T.\ intlst \sqsubseteq \texttt{List}\langle T\rangle$$

which uses quantification over the set of types to predicate that *intlst* has type $\texttt{List}\langle T\rangle$ for some type $T \in \mathcal{J}$.

Up to now, it is not possible to quantify over types in JavaDL. But there are at least two points in the dynamic logic at which such quantification would be of great help.

One situation is when it comes to enumerating all dynamic subtypes of a type. This, as we have seen in section 6.3.1, affects the rule expandDynamicType, but it is also crucial for the process which resolves a method invocation. Though the underlying

type set is infinite, these steps can be expressed in a finite manner if quantification over types is permitted. Specifications on generic classes is the other field in that type quantification is of need. A method contract or a class invariant can be as generic as the class in which it is defined, so that it would be appreciable to be able to proof the correctness of a generic specification *uno pro omnibus* in a generic way, too.

Both situations could be addressed, if formulae like

$$\forall T.\ \phi(T) \qquad \text{and} \qquad \exists T.\ \psi(T)$$

with T ranging over types were possible.

Throughout the next sections the idea of an environment that allows quantification over types is presented. Though going into details at some places, it does not yet cover all eventualities and merely outlines the formal setup.

6.3.4 Formal Fundament

In order to introduce the possibility to quantify over a range of types in the type hierarchy, the fundaments of JavaDL have to be altered, new types and new symbols have to be introduced, and the semantic has to be revisited due to the presence of type variables.

Formally we create an augmented initial partial model $\mathcal{M}^+ = (\mathcal{D}^+, \delta^+, I_0^+)$ which extends the partial model $\mathcal{M} = (\mathcal{D}, \delta, I_0)$ that is the basis for JavaDL (see section 2.2.2.1). Additionally, the type system $\mathcal{T}^+ \supseteq \mathcal{T}$ is augmented by two new types $\mathbb{T}$ and $\mathbb{J}$.

$$\begin{aligned} \mathcal{D}^+ &= \mathcal{D} \overset{\varnothing}{\cup} \mathcal{T} \\ \mathcal{T}^+ &= \mathcal{T} \overset{\varnothing}{\cup} \{\mathbb{T}, \mathbb{J}\} \\ \delta^+(T) &= \begin{cases} \mathbb{J} & \text{if } T \in \mathcal{J} \\ \mathbb{T} & \text{otherwise} \end{cases} \quad \text{for all } T \in \mathcal{T} \end{aligned} \tag{6.5}$$

The domain is enriched by all types of the type system $\mathcal{T}$. Being now elements in the domain, each type $T \in \mathcal{T}$ must be assigned a fixed type in the hierarchy. To represent the types, the two meta types $\mathbb{T}$ and $\mathbb{J}$ have been introduced in the type system $\mathcal{T}^+$. The meta type $\mathbb{T}$ is the "type of all types" and $\mathbb{J} \sqsubseteq \mathbb{T}$ is the meta type of all reference types (see definition 6.1). To avoid the set-theoretically critical situation $\mathbb{T} \in \mathbb{T}$, the metatypes $\mathbb{T}$ and $\mathbb{J}$ have been excluded from the domain $\mathcal{D}^+$. Only $\mathcal{T}$ and not $\mathcal{T}^+$ has been added. The new meta types $\bot \sqsubseteq_0 \mathbb{J} \sqsubseteq_0 \mathbb{T} \sqsubseteq_0 \top$ do not interfere with the other types of the hierarchy, as is depicted in figure 6.4.

Not only the domain and the type system must be modified, the initial signature Σ also experiences some extensions to the sets of function and predicate symbols. For every type $X \in \mathcal{T}$ (or $X \in \mathcal{J}$) that is not a parametrised type, there is a rigid constant function symbol $\mathtt{X} : \mathbb{T} \in \mathrm{FSym}^+$ in the extended set FSym^+ of function symbols (and $\mathtt{X} : \mathbb{J}$ respectively). For every generic class G with n type parameters, the signature contains a rigid function symbol $\mathtt{G} : \mathbb{J}^n \to \mathbb{J} \in \mathrm{FSym}^+$ that takes reference types as arguments.

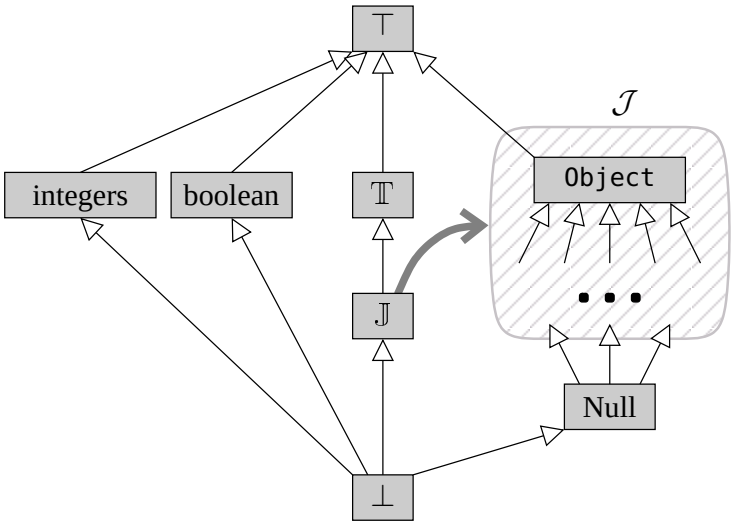

Figure 6.4: The type system augmented by the metatypes $\mathbb{T}$ and $\mathbb{J}$

The symbols have a canonical predefined interpretation which maps them to the corresponding types in $\mathcal{T}$, i.e.

$$\begin{aligned} I_0^+(\texttt{X}) &:= X \in \mathcal{T} \qquad \text{and} \\ I_0^+(\texttt{G})(T_1, ..., T_n) &:= G\langle T_1, ..., T_n\rangle \in \mathcal{T} \end{aligned}$$

There is, for instance, a rigid term `Integer` $\in \mathrm{Trm}_{\mathbb{J}}$ and a rigid term `List<Integer>` $\in \mathrm{Trm}_{\mathbb{J}}$ of type "reference type" with `List` $: \mathbb{J} \to \mathbb{J} \in \mathrm{FSym}^+$ the function which is induced by the generic class List. Parameters to these functions of types are written in angle brackets `<>` just like they are denoted in Java. With these new function symbols, every type in the original type hierarchy is represented by a ground term of type $\mathbb{T}$ in the logic. The introduction of functions that map from types to types unveils the nature of generic classes: They are not types themselves but rather schemata (functions) to which type arguments have to be applied.

With the types available at object level, some predicates can be redefined in a more convenient way. There was for instance one instanceof-predicate $\sqsubseteq T \in \mathrm{PSym}$ per type T. Now it suffices to define one binary predicate[4] $\sqsubseteq : \top \times \mathbb{T} \in \mathrm{PSym}^+$. The repository count constants $count_T :\to \mathrm{Nat} \in \mathrm{FSym}$ can be unified by a function $count : \mathbb{T} \to \mathrm{Nat} \in \mathrm{FSym}^+$, etc.

6.3.5 The Wildcard Symbol

The type parametrisations that contain wildcards have been left out so far. Perhaps the logic could even be constructed without the possibility to explicitly talk about wildcards. It seems achievable that the extension can be performed in such a way that every occurrence of '**?**' can be replaced by a quantified type variable. This is certainly true for some predicates – for instance for the instanceof relation:

$$trm \sqsubseteq G\langle \mathbf{?} \rangle \leftrightarrow \exists T : \mathbb{J}.\ trm \sqsubseteq G\langle T \rangle \tag{6.6}$$

[4] written in infix notation

If all predicates can be expressed in the manner of 6.6, the calculus does not need the wildcard symbol. This reveals also how strongly related the notions of wildcard parametrisation and existential types in type theory are. In the language specification the process of replacing a wildcard with a new type variable is called "capture conversion" ([GJSB05] §5.1.10). The concept has been introduced and described extensively in [THE+04] and [TEH05].

On the other hand, a calculus is suspicious if it does not have a syntactic counterpart for all syntactic elements that can appear in the underlying language. The formal definition from the last section will therefore be once again augmented to support wildcard types.

The new initial model $\mathcal{M}^{++} = (\mathcal{D}^{++}, \delta^{++}, I_0^{++})$ is a refinement of the model $\mathcal{M}^{+}$ proposed in (6.5) and adds one object `wildcard` to the domain and one new type "Wildcard" to the type system. One rigid function symbol **?** : Wildcard is added to the set of function symbols.

$$\begin{aligned} \mathcal{D}^{++} &= \mathcal{D}^{+} \overset{\circ}{\cup} \{\texttt{wildcard}\} \\ \mathcal{T}^{++} &= \mathcal{T}^{+} \overset{\circ}{\cup} \{\text{Wildcard}\} \\ \delta^{++}(\texttt{wildcard}) &= \text{Wildcard} \\ I_0^{++}(\textbf{?} : \text{Wildcard}) &= \texttt{wildcard} \end{aligned} \tag{6.7}$$

The set of functions that have been introduced in the previous section is revised so that for every generic class G with n type parameters a rigid function `G` : $(\mathbb{J} \sqcup \text{Wildcard})^n \to \mathbb{J}$ is available in the set of function symbols FSym^{++}. Consequently generic functions may also be parametrised with '**?**', so that `List<?>` $\in \text{Trm}_{\mathbb{J}}$ is also a valid term which is canonically interpreted as the type `List<?>`$\in \mathcal{T}$ in the original type hierarchy.

This somewhat laborious introduction of the wildcard symbol with a type on its own has been done to ensure that the symbol '**?**' itself does not evaluate to a type and may only be used in type parametrisations and never as a type.

6.3.6 Quantification of Types

With the metatype $\mathbb{T}$ available, it is possible to construct a variable $t : \mathbb{T} \in \text{VSym}$ and syntactically permitted to build a quantified formula $\forall t : \mathbb{T}.\Phi$ or $\exists t : \mathbb{T}.\Phi$ for some formula Φ.

Although it seems that introducing the possibility to quantify over types would imply introducing a higher order into the logic, this is not the case. Variables of types are ordinary variables and types are ordinary elements in the universe of objects. There are predicates and functions with a predefined semantic in the partial model which make them useful.

Yet, the quantification over *types* has implications that are unique to this situation. The following formula that makes use of quantified types shows this specific feature.

$$\forall T : \mathbb{T}.\ \langle \texttt{G<}T\texttt{> g = new G<}T\texttt{>();} \rangle g.value \doteq \text{null}$$

What makes this formula different is that the type T over which is quantified is referred to from within a modality. T is used as a type parameter to a generic class G.

This little example might be a simple method contract for the constructor of class G: "The constructor of G terminates normally for any type parameter T and the attribute *value* is not the null reference afterwards." Such a contract makes sense so that it is favourable to admit type variables as types within modalities. But this includes the new phenomenon that type variables appear inside a Java modality as a type reference.

The rules that can be applied to sequents that contain type variables have to be restricted. Consider for instance the sequent

$$\vdash \exists T\!:\!\mathbb{J}.\langle \texttt{T x = null;}\rangle \,\mathrm{true} \tag{6.8}$$

in which the type variable T appears within a Java program in a quantified context. In traditional KeY, variables are not permitted within modalities. But this cannot be hold up for type variables for various reasons so that they must be dealt with also within code. Hence, if the introduction of the variable was symbolically executed it would result in the sequent

$$\vdash \exists T\!:\!\mathbb{J}.\langle \texttt{x = null;}\rangle \,\mathrm{true} \qquad \text{with } x\!:\!T \in \mathrm{FSym}, \tag{6.9}$$

and a constant x (a program variable) that would have type T – a type variable – would be added to the signature. Being a γ-formula the right hand side of the sequent might be instantiated several times with different terms for T using the rule existsRight. Let A, B be two (non-generic) classes, then two instantiations for T would yield

$$\vdash \{T \leftarrow A\}\langle \texttt{x=null;}\rangle \,\mathrm{true},\ \{T \leftarrow B\}\langle \texttt{x=null;}\rangle \,\mathrm{true},\ \exists T\!:\!\mathbb{J}.\langle \texttt{x = null;}\rangle \,\mathrm{true}\,.$$

In this situation the problem comes to the surface. The type variable is instantiated twice: once with A and once with B. Which would the type of the constant x have to be here? A or B? The problem is that when a variable declaration within a modality is symbolically executed, it triggers the introduction of a new constant function symbol to the set FSym of function symbols in the logic. This new symbol is available also outside the scope of the $\forall T$-quantifier and may therefore lead to problems when T is instantiated.

For that reason the following restriction is to be obeyed by every rule of the calculus:

> The application of any rule must not introduce a term whose type is a type variable, nor may any parameter of the type be a type variable.

This does not make the calculus less powerful, but it limits its performance: γ-formulae (like the one above) are instantiated with ground terms for the quantified variable. At this moment the type variable disappears and the rule, which was inhibited, can now operate normally. The disadvantage is, however, that the simplification cannot be done prior to the instantiation but must be applied after it and so for each instance separately.

Please note that also the role of variables have changed. Unlike "usual" variables that must not appear in modalities, type variables can also be mentioned in modality code. This is because of the lack of an entity corresponding to the program variables for usual variables. It implies that every quantification and every substitution must also be applied to the modality and not only to the involved formulae.

6.3.7 Skolem Types

A quantified formula can also be a δ-formula, thus a universally quantified formula on the right hand side of a sequent or an existentially quantified formula on its left hand side. There are rules (e.g. the rule forallRight) that resolve such contexts by instantiating the variable with a freshly created rigid function symbol (the so-called skolem symbol).

$$\text{forallRight}\ \frac{\vdash\ \{x \leftarrow c\}\phi \quad c{:}T \in \mathrm{FSym}}{\vdash\ \forall x{:}T.\ \phi}$$

Analogously, if a δ-rule is applied to a quantified type variable, it leads to the introduction of a new symbol $Sk{:}\mathbb{J} \in \mathrm{FSym}$.

$$(\text{forallRight})\ \frac{\vdash\ \{T \leftarrow Sk\}\langle \texttt{T x = null;}\rangle\,\mathrm{true}}{\vdash\ \forall T{:}\mathbb{J}.\ \langle \texttt{T x = null;}\rangle\,\mathrm{true}}$$

If, like here, the variable to be substituted also appears within a modality, it has to be changed there, too. The symbol Sk then appears as a type in the Java code. It does *not* introduce a new type however, it merely is a placeholder for an arbitrary type in $\mathcal{J}$. It is not a logical variable (it appears outside the context of any quantifier), so the limitation for instantiation does not hold and symbolic execution may proceed, yielding:

$$(\text{subst})\ \frac{(\text{varDecl})\ \dfrac{\vdash\ \langle \texttt{x = null;}\rangle\,\mathrm{true} \quad x{:}Sk \in \mathrm{FSym}}{\langle Sk\ \texttt{x = null;}\rangle\,\mathrm{true}}}{\vdash\ \{T \leftarrow Sk\}\langle \texttt{t x = null;}\rangle\,\mathrm{true}}$$

Applying the substitution that describes the instantiation to the formula and the program within the modality removes the variable from the formula. The rule varDecl which introduces a new function symbol for a declared program variable may now be applied and introduces a new symbol x which has type Sk – the newly created type symbol.

Such a type Sk is called a **skolem type** for obvious reasons. Skolem types do not introduce new types but are placeholder for arbitrary reference types. That has, of course, consequences on the rules that can be applied upon them.

The relation of Sk to other types w.r.t $\sqsubseteq$ cannot be stated so that the interpretation of terms like $A \sqsubseteq Sk$ does not have a fixed interpretation throughout all structures like ground terms without skolem do. Terms of type $\mathbb{T}$ that do not contain variables and skolem type symbols are rigid ground terms that are fixed by the initial partial interpretation and interpreted identically in any state. Yet, the interpretation of types terms containing skolem symbols can differ from Kripke structure to Kripke structure.

Capturing skolem types formally is a very awkward task. The problem is that the actual type of objects that have a skolem type depends on the interpretation in which a term is evaluated. That a syntactic notion (the type) of a term depends on the interpretation (=semantic) of the expression Sk in a state seems an unhealthy condition.

One solution to overcome the intermingling of syntax and semantic is to allow skolem types as symbols in the logic but not as types for terms. If a variable v of a skolem type Sk is to be introduced, a variable of a supertype is created instead. The premiss $v \in Sk$ is added to ensure proper typing. This way no new type is invented and the typing is still captured – not on the syntactic level of the type hierarchy but on the logic level by a predicate.

$$\mathsf{skolemTypeVarDecl1}\ \frac{v \sqsubseteq Sk \vdash \langle \pi\ \omega \rangle \varphi \quad v\texttt{:Object} \in \mathrm{FSym}}{\vdash \langle \pi\ Sk\ \texttt{v;}\ \omega \rangle \varphi}$$

Here, `Object` is used as the supertype of v because it comprises all possible instantiations of Sk.

This solution has some drawbacks however:

- It creates many new formulae that have to be present in all sequents to appear,
- a static property like typing must be expressed explicitly on the logic level, and
- this approach does not work if the skolem type is a parameter to a generic class since there would not exist a proper supertype.

The alternative that is proposed in the following allows the new type symbol Sk syntactically. But when terms and formula are interpreted, it falls back to a smallest common supertype and thus keeps semantic and syntax separated.

Definition 6.5 (Smallest common supertype) The smallest common supertype $scs(T) \in \mathcal{T}$ for a term $T \in \mathrm{Trm}_{\mathbb{T}}$ (possibly using skolem types) is the smallest type (w.r.t. $\sqsubseteq$) that is supertype to all possible instantiations of T in all interpretations.

For a skolem type Sk or an array over a skolem type $Sk[]^d$ the smallest common supertype is `Object` and `Object[]`^{d} respectively. If T is a parametrised type, then the smallest common supertype is T in which every type argument that contains a skolem type has been replaced by a wildcard. Figure 6.5 lists some examples for smallest common subtypes.

Type T	**Smallest common supertype** $scs(T)$
`Object`	`Object`
Sk	`Object`
$Sk[][]$	`Object`$[][]$
$\mathrm{G}\langle Sk \rangle$	$\mathrm{G}\langle \mathbf{?} \rangle$
$\mathrm{G}\langle G\langle Sk \rangle \rangle$	$\mathrm{G}\langle \mathbf{?} \rangle$
$\mathrm{H}\langle Sk, \texttt{Integer} \rangle$	$\mathrm{H}\langle \mathbf{?}, \texttt{Integer} \rangle$

Figure 6.5: Some smallest common supertypes

The definition of an interpretation has to be adapted to the presence of skolem types. As mentioned above, skolem types should not appear in an interpretation because of their changing configuration. Instead, the smallest common supertypes

are used. An interpretation was defined in definition 2.11 as a function that "lifts" function and predicate symbols to the object level. For terms which have a type that contains a skolem type symbol, the interpretation can only be restricted to the smallest common supertype, which leads to the following modified

Definition 6.6 Let $(\mathcal{D}, \delta)$ be a domain. An interpretation I assigns a function $I(f)$ in the domain to every function symbol $f : T_1 \times \ldots \times T_n \to T \in \mathrm{FSym}$:

$$I(f) : \mathcal{D}_{scs(T_1)} \times \ldots \times \mathcal{D}_{scs(T_n)} \to \mathcal{D}_{scs(T)}$$

An interpretation I assigns a set $I(p)$ of tuples to every predicate symbol $p : T_1 \times \ldots \times T_n \in \mathrm{PSym}$:

$$I(p) \subseteq \mathcal{D}_{scs(T_1)} \times \ldots \times \mathcal{D}_{scs(T_n)}$$

Changing the definition to the smallest common supertype makes sure that the ranges of the functions $I(f)$ and $I(p)$ respectively do not depend on the interpretation of the skolem type symbols. Otherwise the interpretation of a constant $c\!:\!Sk$ would have to read $I(c) : \mathcal{D}_{I(Sk)}$, introducing a strange kind of recursion for I.

With the modified definition a type containing a skolem symbol Sk does not appear as a type on its own at object level, it is replaced by the smallest common supertype. But this implies that an interpretation is not restricted to obey the rules of typing: An interpretation

$$I(c\!:\!Sk) : \mathcal{D}_{scs(Sk)} \text{ with } scs(Sk) = \texttt{Object}$$

can assign to a constant $c\!:\!Sk$ any object because the smallest supertype is `Object`. This would, of course, not be sound any more as in an interpretation I a term that originally holds a reference to the type $I(Sk)$ may then hold any reference type, also one that is not compatible with $I(Sk)$.

To remove such interpretations that are allowed after definition 6.6 but are not compatible with the type system, the interpretations that are to be considered must be confined to those that interpret terms correctly according to the type system.

Definition 6.7 (Compliant interpretation) An interpretation I is compliant with the type system if the following condition holds for any type T and any ground term trm:

$$trm \in \mathrm{Trm}_T \implies \mathrm{val}_I(trm) \in \mathcal{D}_{\mathrm{val}_I(T)}$$

In a given interpretation it is possible to evaluate the term $Sk \in \mathrm{Trm}_{\mathbb{T}}$. The result $\mathrm{val}_I(Sk) \in \mathcal{D}_{\mathbb{T}}$ is by definition a type in $\mathcal{T}$ (since $\mathcal{D}_{\mathbb{T}} = \mathcal{T}$). The index in $\mathcal{D}_{\mathrm{val}_I(Sk)}$ does therefore – though looking exceptional – make sense syntactically.

The set of interpretations that are taken into consideration has already be limited by the introduction of partial initial models which fix parts of the interpretation once and forever. But these confinements are common to all interpretations throughout all structures. The compliance restriction is different as it cannot be formulated by a partial model.

The calculus has temporarily lost its type safety by the modification of the definition of an interpretation. But it will regain it if only compliant interpretations are admitted as valid interpretations in the set of states $\mathcal{S}_\mathcal{K}$ of a Kripke structure $\mathcal{K}$.

Fortunately, compliance with the type system is axiomatisable. The set $\chi \subseteq \mathrm{Fml}$ is valid exactly in those structures in which every state is compliant with the type system.

$$\chi := \{ ''\mathrm{Cl}_\forall(f(x_1, \ldots, x_n) \sqsubseteq T)'' : (f{:}T_1 \times \ldots \times T_n \to T) \in \mathrm{FSym},\ x_i{:}T_i \in \mathrm{VSym}\}$$

The formulae in the set χ make sure that every term is mapped to an object of the domain in a manner that is compliant with the type system. Astonishingly, the notions of an uncompliant interpretation and a state with heap pollution (see 6.1.3 and 6.5) are congruent.

The presence of skolem types require a reconsideration of the definition of the validity of a formula as well.

In section 2.2.2.4 the validity of a formula under an interpretation has been defined. This definition assumes that all types can be considered independent of the interpretation. However, if skolem types are permitted, this is no longer the case, and the validity of a formula has to be adapted in those items dealing with types. This concerns quantifiers and program modalities. Therefore, the definition of the validity of a formula under an interpretation I and a variable assignment β from page 15 has to be revisited in the following points:

$(I,\beta) \models \exists x{:}T.\ \varphi$	iff there is one $d \in \mathcal{D}_{\mathrm{val}_{I,\beta}(T)}$ with $(\mathcal{M}, \beta^d_x) \models \varphi$
$(I,\beta) \models \forall x{:}T.\ \varphi$	iff $(\mathcal{M}, \beta^d_x) \models \varphi$ for all $d \in \mathcal{D}_{\mathrm{val}_{I,\beta}(T)}$
$(I,\beta) \models \langle\pi\rangle\varphi$	iff there is an interpretation I' with $(I, I') \in \varrho(\mathrm{val}_{I,\beta}(\pi))$ and $(I', \beta) \models \varphi$
$(I,\beta) \models [\pi]\varphi$	iff $(I', \beta) \models \varphi$ for all interpretations I' with $(I, I') \in \varrho(\mathrm{val}_{I,\beta}(\pi))$

The quantifiers now use the interpreted type as range over which they quantify, i.e. they take skolem types into consideration and resolve them to the appropriate instantiation that the interpretation assigns them. In program modalities that contain type variables or skolem types, they are replaced by the type instantiation under the interpretation I. That is denoted by $\mathrm{val}_{I,\beta}(\pi)$ for a program π.

Introducing skolem types involves a lot of subtle changes to the semantics of the logic and, hence, is the Achilles heel of the extension of the logic. Syntactical and semantical notions overlap since types – purely syntactical entities – must be treated differently in different states, which is a semantic property.

The setup to include skolem types in JavaDL is only outlined here and its well-definedness cannot be fully guaranteed, but the upcoming sections will show that type metatypes, type quantification and skolem types add new expressivity to the logic and therefore are worth being considered.

6.3.8 Dynamic Subtype Resolution

We have seen in section 6.3.1.2 that the dynamic subtyping rule expandDynamicType has to incorporate all possible dynamic types in order to be a correct rule. With

an infinite set of types, it is in general impossible to give a finite formula that distinguishes between all possible dynamic types of a term.

But now, with the mighty tool of type quantification at hand, the infiniteness of the type hierarchy can be coped with because many dynamic types can be subsumed under one predicate by the use of quantified type variables.

The following observation is essential to the upcoming section as it shows that the relation between any two classes can be expressed in one predicate.

Observation 6.8 Let $G\langle\mathbf{A}\rangle$ be a parametrised instance of the generic class G and H be a generic class.

Then the relationship between $G\langle\mathbf{A}\rangle$ and H is either

- that no instance $H\langle\mathbf{B}\rangle$ of H is a subtype of $G\langle\mathbf{A}\rangle$, or
- that there is a parametrised instance $H\langle\mathbf{B}\rangle \sqsubseteq G\langle\mathbf{A}\rangle$ with type arguments $B_i \in \mathcal{J} \cup \{\mathbf{?}\}$ which is a subtype of $G\langle\mathbf{A}\rangle$, and any other parametrisation $H\langle\mathbf{C}\rangle$ is a subtype of $H\langle\mathbf{B}\rangle$ if and only if it is a subtype of $G\langle\mathbf{A}\rangle$:

$$H\langle\mathbf{C}\rangle \sqsubseteq G\langle\mathbf{A}\rangle \iff H\langle\mathbf{C}\rangle \sqsubseteq H\langle\mathbf{B}\rangle$$

Definition 6.9 $H\langle\mathbf{B}\rangle$ is called the called the **largest subtype of** H for $G\langle\mathbf{A}\rangle$.

$H\langle\mathbf{B}\rangle$ subsumes the relationship between $G\langle\mathbf{A}\rangle$ and all possible instantiations of H in one term. The subtype relationship between a parametrised type $G\langle\mathbf{A}\rangle$ and a class H can therefore be expressed in only one formula if or if not H is a generic class. It is not possible to "pick out" several instances of H as subtypes to $G\langle\mathbf{A}\rangle$ that cannot all be aggregated under one common type.

Proof The following quote can be found in §8.1.5 of the JLS:

> A class may not at the same time be a subtype of two interface types which are different invocations[5] of the same generic interface [...].

The specification disallows for instance for a class C to both implement the interfaces `List<Integer>` and `List<String>` since this would lead to conflicts in the expected types of attributes and methods: Would it be `String get(int)` or `Integer get(int)` that C would have to define?

If the raw type $|H|$ is not a subtype of $|G|$, the first case is true.

If the raw type $|H|$ is a subtype of $|G|$, the relationship can be described in one term. This is because any class can have at most one superclass and every interface may only appear at most once as superinterface (see citation above).

Thus the relation between G and H can always be written in form of a schema like

$$H[V_1, \ldots, V_n] \sqsubseteq G\langle\alpha_1, \ldots, \alpha_m\rangle$$

[5] = parametrisations

with V_j the type variables of H and α_i terms of types that possibly involve the type parameters V_j.

If an instance $G\langle A_1, ..., A_m\rangle$ of the right hand side is given, the schema is applied so that $\alpha_i = A_i$. If the pattern matching cannot be performed, H and $G\langle\mathbf{A}\rangle$ are not related for any parametrisation of H.

But if the pattern can be matched successfully, it may require to instantiate some of the type variables V_j and leave some open. Those that are left open may be instantiated arbitrarily to yield a valid subtype. Hence, an instantiation with '**?**' comprises all possible instantiations and is therefore the sought-after instance. □

Listing 6.11 defines two generic classes for which the largest subtypes shall be given as examples. The largest subtype of D for $C\langle\texttt{Integer}, \texttt{Integer}\rangle$ is $D\langle\texttt{Integer}, \textbf{?}\rangle$. A largest type can thus contain wildcards though the original type does not. The type $C\langle\texttt{Integer}, \texttt{Object}\rangle$ and the class D are completely unrelated on the other hand.

Java 5

```
1 class C<X,Y> {}
2 class D<X,Y> extends C<X,X> {}
```

Java 5 – 6.11

It is possible to put the dynamic typing of a term t for a non-generic class C into a formula

$$t \sqsubseteq_1 C. \tag{6.10}$$

The goal is to put the dynamic typing for a generic class G also in a similar formula – though it actually describes a whole set of types at a time. This is possible because the largest subtype of H subsumes all possible parametrisations in one term. One is tempted to formulate in analogy to (6.10) the relationship as

$$t \sqsubseteq_1 H\langle\mathbf{B}\rangle \tag{6.11}$$

for the largest subtype $H\langle\mathbf{B}\rangle$ of H for the type of t. But this is not correct as $H\langle\mathbf{B}\rangle$ may contain wildcards and then is an abstract type that does not contain objects. What we want to express instead is that any dynamic subtype of the largest subtype $H\langle\mathbf{B}\rangle$ can contain the object t. To do so, a new parametrisation $\mathbf{B}' = (B'_1, \ldots, B'_n)$ is defined w.r.t. $\mathbf{B} = (B_1, \ldots, B_n)$ by

$$B'_i := \begin{cases} V_i & \text{if } B_i = \textbf{?} \\ B_i & \text{otherwise} \end{cases} \tag{6.12}$$

wherein $V_1, ..., V_n : \mathbb{J}$ are new type variables that do not yet appear elsewhere. All wildcards '**?**' in $\mathbf{B}$ are replaced by new type variables in $\mathbf{B}'$. Now the instanceof predicate of t and the class H can be written using the existential type quantification as

$$\exists V_1.\ \exists V_2.\ \ldots \exists V_n.\ t \sqsubseteq_1 H\langle\mathbf{B}'\rangle =: \mathrm{Cl}_\exists\ t \sqsubseteq_1 H\langle\mathbf{B}'\rangle. \tag{6.13}$$

This states that there is a wildcard-free instantiation of H that is a subtype of the type of t and that t is a direct instance of this parametrised type.

With (6.10) and (6.13), formulae are available for both non-generic and generic classes to express dynamic typing and the case distinction that led in section 6.3.1.2 to the inconsistency can now be revised for the use with an infinite type system:

$$\text{expandDynamicTypes}' \ \frac{\begin{array}{ll}\Gamma, & t \Subset_1 C_1 \vee t \Subset_1 C_2 \vee \ldots \vee t \Subset_1 C_n \\ \vee & (\mathrm{Cl}_\exists\, t \Subset_1 G_1\langle \mathbf{B}'_1 \rangle) \vee \ldots \vee (\mathrm{Cl}_\exists\, t \Subset_1 G_m\langle \mathbf{B}'_m \rangle) \\ \vee & t \doteq \mathrm{null} \vdash \Delta\end{array}}{\Gamma \vdash \Delta}$$

with $t \in \mathrm{Trm}_T$ a ground term of a reference type $T \in \mathcal{J}$ appearing in $\Gamma \cup \Delta$ and $C_1, ..., C_n$ the non-generic subtypes of T and $G_1\langle \mathbf{B}'_1 \rangle, ..., G_m\langle \mathbf{B}'_m \rangle$ the largest subtypes for T to the generic classes $G_1, ..., G_m$ that are in a subtype relationship with T.

In this section we have examined how type variables can be used to revise the dynamic type binding rule that suffered severely from the infiniteness of the type system implied by genericity. It is possible to reformulate the dynamic binding rule in such a way that there is only one clause per class. For generic classes existentially quantified type variables are used to obtain this goal.

6.3.9 Method Invocation with Dynamic Types

Method invocation can be treated in two ways in the KeY system: either by symbolically executing the code inside the method body or by applying a method contract that is subject to be proved within another proof. If the object upon which the method is called belongs to a parametrised type, it is vital that the parameters do not include any wildcard type. Both approaches may make use of the type parameters and, hence, they may not be wildcards.

Being an object-oriented language, Java supports the concept of dynamic method dispatch (also called virtual methods), i.e. that the dynamic type of an object decides which implementation of a possibly overridden method is chosen. As in general the dynamic type is not known within verification, a case distinction over all possible method implementations has to be performed.

We will concentrate on methods without arguments and without return value for the purposes of this section as the preparation of parameters can be processed as it is done in the absence of genericity (cf. [BHS07] p. 135ff).

The rule methodCall describes the situation for the call to a method `m()` without arguments that is called on an object *se* where *se* is a simple expression. The classes $C_1, \ldots, C_n$ are the subclasses of the static type of *se* that provide an implementation for the method `m()`. A cascade of nested if-statements decides which implementation has to be chosen.

$$\text{methodCall} \ \frac{\begin{array}{ll}\langle \pi & \texttt{if(se instanceof } C_1\texttt{) \{ se.m()@}C_1\texttt{; \}} \\ & \texttt{else if(se instance } C_2\texttt{) \{ se.m()@}C_2\texttt{; \}} \\ & \ldots \\ & \texttt{else if(se instance } C_n\texttt{) \{ se.m()@}C_n\texttt{; \}}\ \omega\rangle\varphi\end{array}}{\langle \pi\, \texttt{se.m();}\, \omega\rangle\varphi} \tag{6.14}$$

For the implementation of the dynamic method dispatch, it is necessary to choose an implementation of a possibly overwritten class. JavaDL allows therefore statements like `se.m()@C` that are not part of the original Java programming language to denote that the method `m` declared in class `C` shall be called with *se* as the 'this' reference. This way the dynamic method dispatch mechanism is inhibited and the implementation can be explicitly chosen.

This rule will be adapted for the use with generic classes. To change as little as possible, the process of deciding which context to use for the method invocation is divided into two subtasks:

1. *(subtyping polymorphism:)* Decide which implementation (which piece of code) to use to for a method invocation
2. *(parametric polymorphism:)* Decide which parametrisation of the corresponding class has to be taken.

The first is the "old" `methodCall` rule expanded to be used with generic classes as well. The instanceof operator can be used for raw types only but this is sufficient to identify the (generic or non-generic) class that implements the method. The nested if-cascade of (6.14) may then also contain references to generic classes G (without parametrisation) like in

```
if(se instanceof G) { se.m()@G; }.
```

The resulting code may hence contain expressions of the form `se.m()@G` with G being an unparametrised generic class. Yet, the resolution of such calls must be handled differently because if using a contract or if inserting the method body, type parameters play an important role, and they must be instantiated sensibly. Hence, a method call to the raw type must be changed into one with distinct type parameters. The call to `se.m()@G` must therefore be altered to a term that covers all possible instantiations in this context.

To express this, the largest subtype can be used once again. If the dynamic type of *se* is a parametrised version of class G, it is also a subtype of the largest subtype to the type of *se*. Hence, it suffices to take all instantiations that are more specific than the largest subtype into consideration.

Let $G\langle\mathbf{A}\rangle$ be the largest subtype of G to the type of *se* according to definition 6.9. Let further $G\langle\mathbf{A}'\rangle$ denote the largest subtype in which the toplevel wildcards have been replaced by new skolem types $Sk_1, ..., Sk_n$. Then the rule

$$\text{expandGenericCall}\ \frac{se \sqsubseteq G\langle\mathbf{A}'\rangle\ \vdash\ \langle\pi\ \texttt{se.m()@G}\langle\mathbf{A}'\rangle\texttt{;}\ \omega\rangle\varphi}{\vdash\ \langle\pi\ \texttt{se.m()@G;}\ \omega\rangle\varphi}$$

translates a call to a method of an unparametrised generic class to a call to a parametrised version. No wildcards will appear in the parametrisation of the class upon which `m` will be called. While the largest subtype could still contain wildcards, they are now captured by new skolem types. Therefore the rule is especially simple if no wildcards are present in $G\langle\mathbf{A}\rangle$ as the correct subtyping is already ensured by the if-cascade and the additional formula $se \sqsubseteq G\langle\mathbf{A}\rangle$ can be omitted.

$$\text{expandGenericCallSimple}\ \frac{\langle\pi\ \texttt{se.m()@G}\langle\mathbf{A}\rangle\ \omega\rangle\varphi}{\langle\pi\ \texttt{se.m()@G}\ \omega\rangle\varphi}$$

An example shows both the possibilities and the limitations of this access. Listing A.3 shows the implementation of a class `Pair` which is generic and holds two references of the same type. It supports the operation `swap` that exchanges the first and second component of the pair. The following sequent describes that after a call to the method `swap` the first component holds the reference of the second component prior to the call. The call is made on a wildcard instance of Pair.

$$r \doteq q \vdash \{p := q\}\langle \texttt{p.swap();} \rangle r.snd \doteq p.fst$$

with p : `Pair<?>` a program variable for a pair with wildcard parametrisation, q : `Pair<Integer>` a program variable for a pair with a concrete instantiation and r : `Pair<?>` a rigid constant function that represents the reference of p prior to the call.

The dynamic method dispatch applied to this sequent results in the following sequent because there is only one implementation available for the method `swap`: the one in class `Pair`:

$$r \doteq q \vdash \{p := q\}\langle \texttt{p.swap()@Pair;} \rangle r.snd \doteq p.fst$$

The rule expandGenericCall then introduces a new skolem type Sk, because in any case the dynamic type of p must be of a concrete instance of Pair rather than of a wildcard instance, hence yielding

$$r \doteq q, \{p := q\}p \sqsubseteq \texttt{Pair}\langle Sk \rangle \vdash \{p := q\}\langle \texttt{p.swap()@Pair<}Sk\texttt{>;} \rangle r.snd \doteq p.fst.$$

The instanceof predicate $p \sqsubseteq$ `Pair`$\langle Sk \rangle$ has to be prefixed with the same update as the context under which the method call stands. It can thus be simplified to $q \sqsubseteq$ `Pair`$\langle Sk \rangle$ which with the knowledge about the type of $q \sqsubseteq$ `Pair<Integer>` can be used to deduce[6] $Sk \doteq$ `Integer`.

Now the method body may be expanded. A generic equivalent to the non-generic rule methodBodyExpand would result in the following sequent:

$$\begin{array}{ll} & r \doteq q, \{p := q\}p \sqsubseteq \texttt{Pair}\langle Sk \rangle \\ \vdash & \{p := q\}\langle \ \texttt{method-frame(source=Pair<}Sk\texttt{>, this=p) : \{} \\ & \qquad \texttt{}Sk\texttt{ tmp = fst;} \\ & \qquad \texttt{fst = snd;} \\ & \qquad \texttt{snd = tmp;} \\ & \quad \texttt{\}} \ \rangle \ r.snd \doteq p.fst \end{array} \tag{6.15}$$

The method body is enclosed in a "method-frame" statement which is an extension to the Java language grammar added by KeY to express the context in which code stands. The reference *fst* refers for instance to *p.fst* which in the method-frame can be resolved because both the this-pointer (here: p) and the class (here: $Pair\langle Sk \rangle$) are given.

The first statement is a variable declaration which reveals why it is important to capture the wildcard parameter by a new skolem type. The variable *tmp* needs to have a type. This type cannot be definitely stated, but we can use a placeholder – a skolem type – instead.

[6] the rule typesUnique that is defined in the appendix A.3 can be used for this

A symbolic execution of this block is possible and yields the following sequent. Please remember that the updates are parallel so that the reciprocal assignments of *p.fst* and *p.snd* lead to permuted references.

$$\begin{array}{ll} & r \doteq q, \overbrace{\{p := q\}}^{\star} p \sqsubseteq \texttt{Pair}\langle Sk\rangle \\ \vdash & \underbrace{\{p := q\}}_{\star}\{tmp := p.fst \parallel p.fst := p.snd \parallel p.snd := p.fst\} r.snd \doteq p.fst \end{array}$$

The types are consistent so far, *tmp* and *p.fst* are both of type *Sk*. Unfortunately this changes as soon as the updates $\star$ are applied to their respective following formulae, resulting in

$$\begin{array}{ll} & r \doteq q, q \sqsubseteq \texttt{Pair}\langle Sk\rangle \\ \vdash & \{tmp := q.fst \parallel q.fst := q.snd \parallel q.snd := q.fst\} r.snd \doteq p.fst. \end{array}$$

Though this seems to still fit at first glance, problems lie in the detail. The term *p.fst* was an abbreviation for the term $\texttt{Pair}\langle Sk\rangle::fst(p)$. If now p is substituted with q the resulting term $\texttt{Pair}\langle Sk\rangle::fst(q)$ is not well-typed because the function does not expect an argument of type `Pair<Integer>`. But in the antecedent of the sequent it is ensured that q has the appropriate type ($q \sqsubseteq \texttt{Pair}\langle Sk\rangle$) so that a casting operator can be inserted that adjusts the troubling situation to $\texttt{Pair}\langle Sk\rangle::fst((\texttt{Pair}\langle Sk\rangle)q)$.

This short examples unveils both the need for skolem types and the difficulties that they bring along. Without skolem types or a similar anonymous type mechanism, there would be no sensible type for the variable *tmp* in (6.15). On the other hand, a consequence is the "capturing problem", the issue that a type that contains skolem type symbols may or may not be congruent to a concrete type. In the preceding example the type `Pair<Sk>` was captured to become `Pair<Integer>` implying an equality $Sk \doteq \texttt{Integer}$. While the last example allowed to deduce this equality on the logic level, it does not help much on the syntactic level where *Sk* and `Integer` will always remain incompatible. It is possible to insert cast operators where needed since they return the identical object if the types are congruent. But this seems to be an unattractive solution.

The situation is similar to the situation in section 6.3.2, where member of wildcard types were examined: terms that have different static type have to be congruent. While the resolution was possible there, it has to be left open here for the moment.

6.3.10 Static Methods and Fields

Due to the design decision pointed out in section 6.1.1, all parametrised instances of a generic class share the same static context, and references to static members of different parametrisations of the same class refer to the same object. Since type erasure is performed at compile-time, all parametrised instances of a generic class are mapped to the same raw type without type parameters. Therefore every reference to a static member of a parametrised instance is a reference to a static member of the raw type.

Let G be a generic class with one type parameter and a non-static attribute a. If A and B are two different (non-generic) classes, the references `G<A>.a`, `G<B>.a`, and `G.a` refer all to the same location.

The implications on the calculus are minor, however. The mechanism that translates Java locations to terms in JavaDL has to be adapted appropriately. According to definition 6.4 there is only one function symbol $G::a$ in the logic to represent the attribute. It is up to the translation mechanism to map any corresponding expression in the Java language to this function in the logic.

Static methods may not refer to the type variables of the enclosing class. So a call to a static method can be treated like call to a static method in a non-generic class, and the mechanisms of section 6.3.9 do not have to be applied.

Static methods and attributes fit astonishingly well into the existing KeY calculus. Since type erasure is performed by the compiler, there is only one static context per generic class and it does not significantly differ from the static context of a non-generic class.

6.4 Proof Obligations for Method Contracts or Invariants

The goal of formal software verification is to proof that an implementation of a software complies with the specifications made for it. Since Java 5 supports generics, specifications may also be given for a program that makes use of parametric polymorphism. In that case the specification has to be proven for all possible instantiations of the involved type variables.

Specifications in KeY, can be divided into two categories: class invariants and method contracts (see [BHS07] chapter 5 for an introduction). They are given in a formal specification language such as OCL or JML and are translated to JavaDL to obtain corresponding proof obligations.

The Unified Modeling Language (UML), which OCL is part of, does support generic (called parametrised) classes. The OCL specification itself makes use of generic types when defining operations on the generic collections Set and Bag. In the hierarchy of types (OclType), however, there is no explicit mentioning of a class modelling type parameters.

The specification language JML is not explicitly specified to be used in a polymorphic context. Its definitions does not mention type variables nor generic classes. But there is no reason why JML should not be used for generic entities also, and the only amendment that would have to be made involves the appearance of type variables. [Mal06] adapts JML to generic classes for a particular verification system.

Both specification languages are not explicitly equipped with a defined semantic when generic types are used. However, listing 6.12 and the example in section 6.6 show specifications that use type variables and are rather reasonable, so adequate semantic extensions to these languages ought to be thought about.

There is an intuitive definition of a generic specification that does not depend on the language.

Definition 6.10 (Generic specification) Let C be a generic class with n type variables $V_1, \ldots, V_n$ and $\Phi(V_1, \ldots, V_n)$ be a specification for this class: either a

method contract or a class invariant. Then Φ is called a **generic specification** and may depend on the type variables V_i.

Let $C\langle T_1, \ldots, T_n\rangle$ be a parametrised instance of C with $T_i \in \mathcal{J}$. Then the instantiation $\Phi(T_1, \ldots, T_n)$ of the generic specification is a (concrete) specification for $C\langle T_1, \ldots, T_n\rangle$.

Similar to a generic class, a generic specification can thus be considered to be a schema of specification in which the type variables can be instantiated. Note that definition 6.10 does not depend on the specification language in use and applies to OCL, JML, and JavaDL.

In order to prove a generic specification $\Phi(V_1, \ldots, V_n)$, it would be of great help if the proof could be performed for all instantiations at the same time. If type variables were not present in the calculus, a specification would have to be proven for every instantiation separately. We have seen in previous sections that the type system has to be infinite if generic classes are allowed which implies that one cannot compute all needed instantiations in advance. Proofing (quasi) the same obligation more than once can be avoided by type quantification.

Section 6.3.6 has laid the formal fundaments for type quantification that are used in the following. If Φ_{JDL} is the translation of a generic specification Φ to JavaDL, its proof obligation is the universal closure over all type variables:

$$\vdash \forall V_1 : \mathbb{J}.\ \forall V_1 : \mathbb{J}.\ \ldots \forall V_n : \mathbb{J}.\ \Phi_{\text{JDL}}(V_1, \ldots, V_n) \tag{6.16}$$

The declaration of the generic class C in listing 6.12 for example contains two generic specifications Φ_1 and Φ_2 which are method contracts for the methods `m1` and `m2` formulated in JML. Φ_2 is a specification that literally mentions the type variable T.

Java 5 with jml

```
1  class C<T> {
2
3      T value;
4
5      /*@ public normal_behavior Φ1
6        @   requires t != null;
7        @   ensures value != null;
8        @*/
9      void m1(T t) {
10         value = t;
11     }
12
13     /*@ public normal_behavior Φ2
14       @   requires object instanceof T;
15       @*/
16     void m2(Object object) {
17         value = (T)object;
18     }
19 }
```

Java 5 with jml – 6.12

The generic proof obligation after 6.16 for the method contract Φ_1 of the first method `C.m1()` translated to JavaDL reads

$$\vdash \forall T:\mathbb{J}.\ \forall c:C\langle T\rangle.\ \forall t:T.\ \big(\psi \rightarrow \underbrace{t \doteq \text{null}}_{\text{requires}} \rightarrow \langle \texttt{c.set(t);}\rangle(\underbrace{c.value \doteq \text{null}}_{\text{ensures}})\big) \tag{6.17}$$

with ψ a formula that captures some preconditions on c and t. The quantified variable $t:T$ is a variable whose type T is a quantified variable also. In section 6.3.6 we have seen that a term may not have a type that is variable itself, so that $t:T$ cannot be allowed. It is hence not possible to formulate the proof obligation this way. But the formula (6.17) is a δ-formula valid if and only if the equation

$$\vdash \forall c:C\langle Sk\rangle.\ \forall t:Sk.\ \big(\psi' \rightarrow \underbrace{t \doteq \text{null}}_{\text{requires}} \rightarrow \langle \texttt{c.set(t);}\rangle(\underbrace{c.value \doteq \text{null}}_{\text{ensures}})\big) \tag{6.18}$$

is valid. (6.18) emerges from (6.17) by instantiating the outer-most quantified variable T with a fresh skolem symbol Sk. Remember that it is allowed to create variables and function symbols of skolem types.

The method contract Φ_2 for `C.m2()` in listing 6.12 shows that it makes sense to allow generic types also in specifications. While the expression `object instanceof T` would not be valid in the Java language, it ought be allowed in the superset of JML that is used with generic classes. Using the generic precondition, the subsequent cast to a type variable can be proven to be correct.

In this section we have seen that specification languages can be canonically extended to support genericity. Type variables, skolem types, and quantification over types are vital to proof specifications in a generic context.

6.5 Unchecked Operations

Java is a statically typed languages, i.e. every expression e has got a type that can be deduced at compile-time and which is called the *static type* S of the expression. Because of the subtyping polymorphism (in particular the inheritance relationship between classes and interfaces) it is possible that at run-time an expression e has got a *dynamic type* D that differs from the static type T but is a subtype of it (i.e. $D \sqsubseteq T$).

The invariant $D(e) \sqsubseteq S(e)$ that for any expression e in a program the dynamic type is a subtype of the static type is called *type safety*[7]. The process of violating this principle is called *heap pollution*. Traditional Java is typesafe under any circumstances, type safety has even been proven for parts of the language (cf. [IPW99]). In a tutorial [Bra04] for Java generics, the author Bracha, who is a co-designer of Java's generic facilities, states

> that if your entire application has been compiled without unchecked warnings using `javac -source 1.5`, it is type safe.

[7] There exist other definitions of the notion "type safety" that are quite different to this. This is a rather relaxed one that fits best to the needs for this context.

Conversely this suggests that type safety is not guaranteed if there *are* unchecked warnings present. And this is the case. We have seen in 6.1.3 that a cast that contains a type variable is unchecked and can therefore lead to a polluted heap.

The pollution can happen because Java is not a purely statically typed language as has been pointed out in section 6.1.1. There are expressions that need type information at run-time. But only some type information is available due to fact that type erasure is performed on parametrised types and type variables. The operation that suffers from this deficiency is the explicit cast. All expression that cause an unchecked warning are expression that cast to a type of which some or all information is lost.

Two kinds of unchecked casts can be distinguished:

Cast to a type variable If E is a type variable and *exp* an expression with a static type compatible with E, the expression `(E)exp` is an unchecked cast because it cannot be checked at run-time.

Cast to a parametrised type If $G\langle\mathbf{A}\rangle$ is a parametrised type, that may (but does not have to) contain type variables and *exp* an expression whose static type is a supertype to $G\langle\mathbf{A}\rangle$, the expression `(G`$\langle\mathbf{A}\rangle$`)exp` is an unchecked cast, because it cannot be checked at run-time.

Unchecked casts are *completely unchecked* if the type information that is available at run-time does not suffice to perform any type checking. Unchecked casts are *partially unchecked* if the information available at run-time allows to perform type checking to a certain degree. Without bounded type variables, casts to type variables can only be completely unchecked.

Unchecked casts have to be handled by symbolic execution with care. The execution must ensure an unpolluted environment, the expressions that are to be cast need to be enforced to have a compatible type.

This leads to a refinement to the description of programs that KeY can deal with: KeY works on correct Java programs that are guaranteed to not pollute the heap. It is thereby ensured that type safety is never lost neither on Java variables nor on terms in the logic. This refinement is not really a restriction since the state of a polluted heap is considered both undesired and faulty.

Let us first have a look at the rule that handles explicit casts for reference types in traditional JavaDL for a reference type *type* and a simple expression *se*:

$$\text{referenceCast}\ \frac{\langle\pi\ \texttt{if (}se\texttt{ == null) }lhs\texttt{ = null; else else if (}se\texttt{ instanceof }type\texttt{) }lhs\texttt{ = }se\texttt{; else throw new ClassCastException(); }\omega\rangle\varphi}{\langle\pi\ lhs\texttt{ = (}type\texttt{)}se\texttt{;}\omega\rangle\varphi}$$

Basically referenceCast is a distinction between two cases: either *se* is of type *type* (or null), so that the assignment will be performed, or it is not in which case an exception is thrown. Execution continues in either case. For a (partially) unchecked cast (*type*)*se* there are three general cases to be distinguished:

1. **The congruent part:** *se* has a type that is a subtype of *type* ($se \sqsubseteq type$) in which case the assignment may be made and the program be continued as type safety is guaranteed.

2. **The checked part:** *se* has a type that is not a type variable and whose class is incompatible with the unparametrised version of *type*. This is the "partial" part of a cast that can be checked, so that an `ClassCastException` will be thrown even though this is an unchecked cast.

3. **The unchecked part:** *se* has a type that either is a type variable or the class of the type is a subclass to the class of *type* but of a different parametrisation. The parametrisation cannot be checked at run-time so that throwing a `ClassCastException` would not match the behaviour of Java. Yet, execution must not proceed as it would entail heap pollution.

The operational semantic of the first two cases can be described as it has been done up to now. For the third case the operational semantic in the logic is still to be defined. Symbolic execution of the cast operation may not lead to a next program state then. This is expressed in terms of the dynamic logic as the fact that in this case the modality does not lead to a successor world in the Kripke structure. This is also the case for a program that terminates abruptly. The simplest program that terminates abruptly consists of a single `break`-statement only so that this will be used to describe the semantic at this point. A failure of an unchecked cast can therefore be compared with the occurrence of an exception that cannot be caught.

Let us look at an example that demonstrates the three cases. In listing 6.13 the classes G, A, and B are defined. Method `Example.m()` contains examples for all three possible types of casting contexts.

Before we can formulate the rules for the calculus dealing with unchecked casts, the question when a cast is unchecked remains to be answered. It is unchecked if the type that is cast to is a type variable or a parametrised type **originally**. "Originally" is of importance here since a type variable may be instantiated with a concrete type during the process of symbolic execution and thus turn an actually unchecked cast into one which seems to be checked.

Java 5

```
class A    { }
class B    { }
class G<T> {
    T m(Object o) { return (T)o; }
}
```

Java 5 – 6.14

With the small generic class C defined in listing 6.14, it can be of interest to try to proof the formula

$$\langle \texttt{C<A> c = new C<A>(); B o = new B(); c.m(o);} \rangle \, \text{true} \tag{6.19}$$

which is a test of termination for the method `c.m` under certain circumstances. If the formula $\langle p \rangle \, \text{true}$ can be proven for a program p then p terminates normally

Java 5

```
1  class G<X> { }
2  class A    { }
3  class B    { }
4
5  class Example {
6
7      static void m() {
8          Object object;
9          G<A> g;
10
11         // case 1: congruent part
12         // The variable object has an appropriate type, so this is good.
13         object = new G<A>();
14         g = (G<A>)object;
15
16         // case 2: checked part
17         // The type of object and G⟨A⟩ are not compatible:
18         // An exception will be raised.
19         object = new Object();
20         g = (G<A>)object;
21
22         // case 3: unchecked part
23         // The type of object G⟨B⟩ and G⟨A⟩ differ only in parametrisation.
24         // This would lead to heap pollution is therefore to be prevented.
25         object = new G<B>();
26         g = (G<A>)object;
27     }
28 }
```

with an unpolluted heap. The latter is because polluting programs have been given the semantic of abruptly terminating programs. Without going into details about the object creation and method invocation process, the formula from (6.19) will be transformed to a context similar to the following in the process of symbolic execution:

$$\langle \texttt{method-frame(class=Cast<A>, this=c):\{ return (A)o; \}} \rangle \, \text{true} \tag{6.20}$$

All of a sudden and due to the instantiation of the type variable T by the class `A`, the unchecked cast `(T)o` has gone and been replaced by a checked cast `(A)o`. Continuing from that point on would not be correct: The method `C.m()` does never throw a `ClassCastException`, but the symbolic execution to follow would.

Therefore all explicit casts that convert to a type variable (or an array over a type variable) have to be marked when they are encountered during the semantic analysis of programs or code fragments used in modalities. I propose that such casts will be denoted with double parentheses like in `((E))exp`. If type variables get instantiated later (e.g. by method invocation or instantiation of a γ-formula, etc.) the cast is still tagged as an unchecked one and will be treated as a such even though the type variable may have gone.

Everything that is needed has been defined so that now the rules that deal with unchecked casts can be introduced. They will appear in pairs: One for the $\langle\cdot\rangle$-modality and one for the $[\cdot]$- modality which is slightly different. The reason for this is that the program that does not lead to a succeeding state (the break-operator) has different semantics in the modalities:

$$\langle \texttt{break;} \rangle \phi \leftrightarrow \text{false} \qquad \text{while} \qquad [\texttt{break;}] \phi \leftrightarrow \text{true} \qquad \text{for any formula } \phi.$$

The rules may not be defined independently, they are related due the modal tautology $\Box\phi \leftrightarrow \neg\Diamond\neg\phi$ with which they have to comply. Note that the Java `instanceof`-operator cannot be used like in the traditional rule referenceCase as it may only operate on non-parametrised types that are not type variables. But *type* may be a type variable so that the instanceof-operator of JavaDL has to be used.

$$\text{typeVarCastDiamond} \; \frac{se \sqsubseteq type \wedge \langle \pi \; lhs \; \texttt{=} \; se \texttt{;} \; \omega \rangle \varphi}{\langle \pi \; lhs \; \texttt{=} \; \texttt{((}type\texttt{))}se \texttt{;} \; \omega \rangle \varphi}$$

$$\text{typeVarCastBox} \; \frac{se \sqsubseteq type \rightarrow [\pi \; lhs \; \texttt{=} \; se \texttt{;} \; \omega] \varphi}{[\pi \; lhs \; \texttt{=} \; \texttt{((}type\texttt{))}se \texttt{;} \; \omega] \varphi}$$

In these rules the case discussed distinction seems to have vanished. The antecedents are the result of a propositional simplification of the longer formulae performing the distinction for the completely unchecked case:

$$\begin{aligned} & (se \sqsubseteq type \rightarrow \langle \pi \; lhs \; \texttt{=} \; se \texttt{;} \; \omega \rangle \varphi) \\ \wedge \quad & (se \not\sqsubseteq type \rightarrow \underbrace{\langle \texttt{break;} \rangle \varphi}_{\text{false}}) \end{aligned} \tag{6.21}$$

$$\begin{aligned} & (se \sqsubseteq type \rightarrow [\pi \; lhs \; \texttt{=} \; se \texttt{;} \; \omega] \varphi) \\ \wedge \quad & (se \not\sqsubseteq type \rightarrow \underbrace{[\texttt{break;}] \varphi}_{\text{true}}) \end{aligned} \tag{6.22}$$

The rules dealing with partially unchecked casts follow the same schema and look rather similar with the extension that there is an additional distinction between the checked and the unchecked case.

$$\text{partialCastDiamond}\ \frac{\begin{array}{l}\text{if } se \sqsubseteq G\langle ?, \ldots, ? \rangle \\ \quad \text{then } se \sqsubseteq G\langle \mathbf{A} \rangle \wedge \langle \pi\ lhs\ \texttt{=}\ se\texttt{;}\ \omega \rangle \varphi \\ \quad \text{else } \langle \pi\ \texttt{throw new ClassCastException();}\ \omega \rangle \varphi\end{array}}{\langle \pi\ lhs\ \texttt{=}\ \texttt{((}G\langle A \rangle\texttt{))}se\texttt{;}\ \omega \rangle \varphi}$$

$$\text{partialCastBox}\ \frac{\begin{array}{l}\text{if } se \sqsubseteq G\langle ?, \ldots, ? \rangle \\ \quad \text{then } se \sqsubseteq G\langle \mathbf{A} \rangle \rightarrow [\pi\ lhs\ \texttt{=}\ se\texttt{;}\ \omega] \varphi \\ \quad \text{else } [\pi\ \texttt{throw new ClassCastException();}\ \omega] \varphi\end{array}}{[\pi\ lhs\ \texttt{=}\ \texttt{(}G\langle A \rangle\texttt{)}se\texttt{;}\ \omega] \varphi}$$

In this section we have developed rules to handle cast operations that are unchecked due to Java's deficiencies. We have provided a complete operational semantic that also describes the semantic in case of heap pollution. A different approach to the problem is to underspecify the situation and provide operational rules only for the cases that do not harm type safety. The remainder is left open and no more rules can be applied. The calculus would be less complete, but only the controversial points would be not fixed and left open. One according rule would be

$$\text{typeVarCastDiamondIncomplete}\ \frac{\begin{array}{lll} se \sqsubseteq type & \vdash & \langle \pi\ lhs\ \texttt{=}\ se\texttt{;}\ \omega \rangle \varphi \\ & \vdash & se \sqsubseteq type \end{array}}{\vdash \quad \langle \pi\ lhs\ \texttt{=}\ \texttt{((}type\texttt{))}se\texttt{;}\ \omega \rangle \varphi}$$

6.6 Generic Arrays – an Example

One major drawback of the often cited design decision is that arrays over type variables cannot be created. If T is a type variable then one would intuitively expect that a statement like

```
T[] array = new T[10];
```
(6.23)

could be used to created an array which holds elements of type T. However, this statement is not possible in Java since the information about the actual instantiation of T is not present at run-time.

But creating an array over a generic type variable is a reasonable task. Generic classes are very often used for lists, sets, maps, or other collection types which relying on arrays in their implementations. A prominent example is the class `java.util.ArrayList` which implements the API for a list by means of arrays. The reference implementation uses an assignment involving an unchecked conversion similar to the following to create an array over a type variable[8]:

```
T[] array = (T[])new Object[10];
```
(6.24)

An array with entries of type `Object` is created and then explicitly cast to an array with entries of type T. If type information was present at run-time, this would fail

[8] see the implementation of java.util.ArrayList

under any circumstances. But with the type information not available, this statement definitely introduces heap pollution: The location *array* contains a reference to a type (`Object[]`) that is not compatible with the static type of *array* (`T[]`) unless T is instantiated with `Object`.

In the Java Collection Framework, these cases of heap pollution are well-calculated, and it is always ensured in the context of such arrays that only elements of compatible types are stored.

The formal verification, however, cannot tolerate such a situation. The malicious statement

```
((Object[])array)[0] = new Object();
```

introducing a ill-typed expression into the array should give raise to an `ArrayStoreException`. It does not for an array which was created like in statement 6.24 in which case an additional proof goal would have to be opened.

In a general context it is not possible to always distinguish between a 'good' array of a certain type and one at whose creation heap pollution had been introduced. The policy of disallowing heap pollution at any rate may not be dropped.

This raises the question how generic arrays are to be implemented if statement 6.23 is invalid and 6.24 leads to heap pollution. The answer is somewhat disappointing: Not at all. Instead use an array that holds arbitrary objects and thus cannot introduce heap pollution. In the presence of a formal verifier, the condition that an array a only holds elements of a type T can be framed in a formula $\theta(a, T)$:

$$\theta(a, T) := \forall i : \text{Nat}.\ (i < a.length \rightarrow a[i] \sqsubseteq T) \tag{6.25}$$

If the property $\theta(a, T)$ is ensured, any access to an array element $a[i]$ for any i within the valid range is save and does neither provoke an exception nor possibly pollute the heap. The following formula which ensures that the cast terminates normally and does not pollute the heap can be proven:

$$\forall i : \text{Nat}.\ \forall T : \mathbb{J}.\ \big((a.\texttt{<created>} \doteq \text{TRUE} \wedge a \neq \text{null} \wedge i < a.\texttt{length} \wedge \theta(a, T)) \rightarrow \langle \texttt{T elem = ((T))a[i];} \rangle \text{true}\big)$$

Thus it must be guaranteed at the level of the logic that the array a only holds references to objects of type T rather than on the level of static or dynamic type checking.

If the array a is wrapped into a generic class declaration $Array\langle T \rangle$ and the condition $\theta(a, T)$ is used as a class (instance) invariant, typesafe read and write access functions can be implemented for a.

In appendix A.3 you will find an implementation of the class `Array<T>` in listing A.4. The source code is enriched with comments that contain JML specifications. The invariant $\theta(a, T)$ appears in lines 7 and 8. The one problematic statement is the cast operation in line 24 which is an unchecked cast. The method `get` that surrounds this statement has a contract formulated in JML that translates to JavaDL basically as

$$\begin{aligned}&\forall a : \texttt{Array}\langle Sk \rangle.\ \forall i : \text{Int}.\\ &\quad \big((a.\texttt{<created>} \wedge a \neq \text{null} \wedge 0 \leq i < a.array.\texttt{length} \wedge\\ &\qquad \theta(a.array, Sk) \wedge a.array \neq \text{null}) \rightarrow \langle \texttt{r = a.get(i)} \rangle (r \doteq a.array[i])\big)\end{aligned}$$

with Sk a new skolem type and $r : Sk$ a program variable of type Sk.

The class Array allocates an array of arbitrary objects and converts them to the type parameter only when there is a reading access (line 24). The unchecked conversion that performs the necessary cast can be made because the class invariant ensures that the array contains only elements of the needed type. This invariant has to hold before and after each invocation of any constructor or non-static method.

The generic class `Array` can be used at any place in a program when an array over a generic type is to be used. It does not pollute the heap and has properties that are well-defined and can be reviewed by a verifier that is aware of generics.

This example has demonstrated that the augmented KeY calculus can be used to prove the absence of heap polluting statements even if unchecked operations have to be used. The method contract of the method `put` utilises a type variable in an instanceof operation. This strongly suggests that a generic extension of JML should support a superset of the expressions allowed in Java.

6.7 Conclusion

Generics are a challenge for KeY. They do not fit into the existing framework without considerable effort. Discarding them by a statical analysis is a possible, yet unsatisfactory way to treat them. Hence, parametric polymorphism ought to be devolved on the logic as well. This transfer, however, implies revising several fundamental notions which includes rules that may no longer applied, an infinite type system, and other unexpected changes.

Quantification over types can be transferred from other calculi dealing with parametric polymorphism to JavaDL. Together with the introduction of type meta types and skolem types, it provides a solution to some of the above problems. But the new concepts in the logic bring along that some basic principles of the underlying logic have to be reconsidered. Especially skolem types impose an interdependence between syntactic and semantic notions that is difficult to seize. The problems that appeared with skolem types during method dispatched reveal that the type system that KeY relies upon is not flexible enough to deal with the notion of a type variable.

Yet, despite the absence of an unassailable formal fundament, type variables and quantification are welcome extensions to the logic that allow to talk about a variety of topics that could not be addressed otherwise. It is due to them that dynamic type distinction and method dispatch can be performed in a generic context. Section 6.6 demonstrates that the verification is of great use to prove the absence of heap pollution.

It is worth a more detailed study whether the new expressivity which is needed to deal with parametric polymorphism does imply problems on the logic. It is still a long way to go before a sound and complete extension of the logic and the calculus can be presented for generic classes.

7. Summary and Future Prospects

Summary

The Java programming language has been extended by several new concepts and constructs in release 5. In this thesis, I have brought together with KeY four of these features, examining their implications on the verification tool. The thesis comprises the analysis of enumerated types, enhanced for loops, autoboxing, and generics.

What all considered novelties have in common is that their applications can be replaced by code in traditional Java in a natural manner. This is not unexpected if one takes into account that Java 5 has been designed to be downward compatible to allow interoperability with existing code.

It would thus be possible to remove all traces of Java 5 from a program yielding a program in traditional Java. But this would exclude any benefits that the innovations might have from the verification process. Instead, every new feature has been handled individually.

Typesafe enumerations add a new keyword 'enum' to the language that can – like 'class' – be used to declare a new datatype. In fact, enums are usual classes that have certain invariant properties guaranteed by the compiler and the language. The advantages are obvious: The invariants may be used for the verification of specifications, and yet, they themselves do not have to be proven.

These properties capture that enumeration datatypes have a fixed number of instances and can be expressed in first order logic. Because of the way in which KeY models the heap, the rules that express the properties can be chosen very elementary. Enums seamlessly fit into the system.

Enhanced for loops allow to traverse an array or a collection of objects in a more intuitive way than traditional for loops. Though a transformation to traditional loops can be performed very easily, this is not the only way to deal with enhanced loops. If an array is subject to an iteration in an enhanced for loop and the loop body terminates in each iteration, the whole loop is guaranteed to terminate also

– in contrast to the traditional for loop in Java for which termination cannot be guaranteed even if the loop body terminates.

By means of an invariant rule that has been adapted to the language extension, it is possible to shorten proofs on loops considerably, a sample proof has been shown that only needed a third of the size of a similar proof with a traditional while loop.

To bring primitive and reference datatypes closer together, **autoboxing** and **auto-unboxing** have been introduced in Java 5 making it is possible to use primitive values in places where references to objects are expected and vice versa. The translation is done implicitly by conversions.

It is possible (though laborious) to identify boxing and unboxing contexts in the calculus and to mark these locations. A set of rules has been set up that models boxing and unboxing conversions in the dynamic logic.

Generics introduce parametric polymorphism into the Java language. They are by far the novelty with the most grave and most important impact. Again, the reduction to traditional Java is possible in a natural fashion. If, however, the new facilities are to be supported by the logic as well, there are several points at which it has to be severely changed.

The type system cannot any longer be guaranteed to be finite because of the possibility to create arbitrarily deeply nested types. A type system that grows over the time has been proven to be untenable since it would destroy the constant domain paradigm and introduce unsoundness.

From the polymorphic types lambda calculus, the idea of existential and universal types has been adapted to JavaDL and formally introduced. Due to the structure of JavaDL as a typed predicate logic, it has proven helpful to introduce new type symbols, the *skolem types*, which are placeholders for arbitrary Java reference types.

With quantification over types and skolem types, generic specification can be handled as well as dynamic type distinction and dynamic method dispatch. Some details had to be left open due to the limited time available for this thesis.

In the introduction, two aspects of an examination of Java 5 were mentioned: the adaptability of KeY and the formal capability of the new language extensions. As far as the flexibility is concerned, KeY has proven to be able to cope with all innovative structures. However, when establishing generics, the system has turned out to be a little inflexible because of its strict typing system.

All new language elements do have their right of existence from the point of view of formal verification. Most of them implicitly carry valuable additional information that can be brought into the process. All features imply at least minor potentials for typical programming errors, and KeY can be used to reduce their thread.

Future Prospects

Enumerated types, enhanced for loops, and autoboxing fit very well into the underlying concepts of KeY. This thesis provides proposals how they can be integrated into the formal verification system. Implementations have been produced or outlined. Therefore these fields leave only very little space for further studies.

Generic polymorphism in Java, on the other hand, still allows exhaustive research. The extension of the calculus could only be outlined within this thesis and the introduced formal fundament would require further consolidation.

To fully support generics in the calculus, an extensive set of new rules would to have to be set up and implemented. The impacts of parametric polymorphism on the efficiency of the KeY system and the implications on automatic proofing should also be subject to further studies.

A. Java Programs

A.1 Enhanced For Loops

An enhanced for loop can be used to traverse a collection of objects. Unfortunately it cannot be ensured that the run time of such an iteration is finite. The following implementation of a linerly-linked list `Chain` demonstrates that collections that permit appending during traversal can lead to diverging programs.

This example has been used in section 4.3.3.

JAVA 5

```
1
2  class Chain implements Iterable {
3
4     Object head;
5     Chain tail;
6
7     public Chain(Object object) {
8        tail = null;
9        head = object;
10    }
11
12    public void append(Object object) {
13       if(tail != null)
14          tail.append(object);
15       else
16          tail = new Chain(object);
17    }
18
19    public ChainIterator iterator() {
20       return new ChainIterator(this);
21    }
22
23 }
24
25 class ChainIterator implements java.util.Iterator {
26
```

```
27    private Chain chain;
28
29    ChainIterator(Chain chain) {
30       this.chain = chain;
31    }
32
33    public boolean hasNext() {
34       return chain != null;
35    }
36
37    public Object next() {
38       Object ret = chain.head;
39       chain = chain.tail;
40       return ret;
41    }
42
43    public void remove() {
44       throw new UnsupportedOperationException();
45    }
46 }
47
48 public class ChainProblem {
49    public static void main(String args[]) {
50       Chain chain = new Chain(new Object());
51       chain.append(new Object());
52
53       //
54       // Loop 1: simple traversal
55       //
56       for(Object t : chain) {
57          System.out.println(t);
58       }
59
60       //
61       // Loop 2: traverse and append
62       //
63       for(Object t : chain) {
64          System.out.println(t);
65          chain.append(new Object());
66       }
67    }
68 }
```

A.2 Autoboxing and -unboxing

The class `java.lang.Integer` is part of the core package of the Java programming language. To implement the rules for autoboxing and auto-unboxing, the calculus has to assume details of an implementation. The following listing depicts the relevant parts of the class used internally within KeY. In particular, the attribute `value` storing the boxed primitive value is of interest.

Section 5.2.3 has made use of this definition.

— Java 5 —

```
1  package java.lang;
2
3  public final class Integer extends Number implements Comparable<Integer> {
4
5      private int value;
6
7      public Integer(int value) {
8          this.value = value;
9      }
10
11     public int intValue() {
12         return value;
13     }
14
15
16
17     private static class IntegerCache {
18         private IntegerCache(){}
19
20         static final Integer cache[] = new Integer[-(-128) + 127 + 1];
21
22         static {
23             for(int i = 0; i < cache.length; i++)
24                 cache[i] = new Integer(i - 128);
25         }
26     }
27
28     public static Integer valueOf(int i) {
29         final int offset = 128;
30         if (i >= -128 && i <= 127) {
31             return IntegerCache.cache[i + offset];
32         }
33         return new Integer(i);
34     }
35
36
37     //
38     //  . . .
39     //
40
41 }
```

A.3 Generics

Generic Pairs

The following generic class `Pair<T>` has been mentioned in section 6.3.9 revealing the problems that occur when skolem types are captured. The variable `tmp` (in the method `swap`) whose type is type variable is used to illustrate this aspect in an example.

Java 5 with JML

```
1  class Pair<E> {
2
3      private E fst;
4      private E snd;
5
6      public Pair(E fst, E snd) {
7          this.fst = fst;
8          this.snd = snd;
9      }
10
11     public void swap() {
12         E tmp = fst;     // a local variable of type E − a type variable
13         fst = snd;
14         snd = tmp;
15     }
16
17     public E getFst() { return fst; }
18
19     public E getSnd() { return snd; }
20
21 }
```

Java 5 with JML – A.3

Disjoint Types

If a term x is a member of two different parametrisations $G\langle\mathbf{A}\rangle$ and $G\langle\mathbf{B}\rangle$ of the same generic class G, where $\mathbf{A}$ and $\mathbf{B}$ do not contain wildcards at toplevel, both instances must be the same. Two different wildcard-free instantiations have no objects in common: $\mathbf{A} \neq \mathbf{B} \implies G\langle\mathbf{A}\rangle \sqcap G\langle\mathbf{B}\rangle = \bot$

This induces the following rule that allows to deduce the equality of type parameters.

$$\mathsf{typesUnique}\ \frac{A_1 \doteq B_1,\ \ldots, A_n \doteq B_n,\ x \sqsubseteq G\langle\mathbf{A}\rangle,\ x \sqsubseteq G\langle\mathbf{B}\rangle\ \vdash}{x \sqsubseteq G\langle\mathbf{A}\rangle, x \sqsubseteq G\langle\mathbf{B}\rangle\ \vdash}$$

Generic Arrays

The generic extension in Java 5 does not allow to create arrays over parametrised types and type variables. The class `Array<T>` can be employed to construct typesafe arrays over any type.

It has been used ins section 6.6 to demonstrate that an augmented calculus can ensure the absence of heap pollution.

Specification are stated in JML within annotating comments in the class.

JAVA 5

```
1  public class Array<T> {
2
3      private Object array[];
4
5      /*@ invariant
6        @    array != null &&
7        @    (\forall int i; 0 <= i && i < array.length;
8        @             array[i] instanceof T || array[i] == null);
9        @*/
10
11     /*@ public normal_behavior
12       @    requires size >= 0;
13       @    ensures array.length == size;
14       @*/
15     public Array(int size) {
16         array = new Object[size];
17     }
18
19     /*@ public normal_behavior
20       @    requires 0 <= i && i < array.length;
21       @    ensures \result == array[i];
22       @*/
23     public T /*@ pure @*/ get(int i) {
24         return (T)array[i];
25     }
26
27     /*@ public normal_behavior
28       @    requires 0 <= i && i < array.length;
29       @    ensures array[i] == t;
30       @*/
31     public void set(int i, T t) {
32         array[i] = t;
33     }
34
35     /*@ public normal_behavior
36       @    requires true;
37       @    ensures \result == array.length;
38       @*/
39     public /*@ pure @*/ int length() {
40         return array.length;
41     }
42
43 }
```

Bibliography

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 49–65, New York, NY, USA, 1997. ACM Press.

[BB04] Gilad Bracha and Joshua Bloch. Extending the Java Programming Language with enumerations, autoboxing, enhanced for loops and static import. http://jcp.org/en/detail?id=201, February 2004.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.

[Blo01] Joshua Bloch. *Effective Java Programming Language Guide*. Prentice Hall PTR, 1st edition, June 2001.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 46–69. Springer-Verlag, 2005.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[Bra04] Gilad Bracha. Generics in the java programming language, July 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf DL: March 12, 2007.

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[Car97] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[CG98] Robert Cartwright and J. Guy. Compatible genericity with run-time types for the Java Programming Language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–215. ACM Press, 1998.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Gaf06] Neal Gafter. Reified generics for java. Neal Gafter's blog: Thoughts about the future of the Java Programming Language, 5th November 2006. http://gafter.blogspot.com/2006/11/reified-generics-for-java.html DL: March 12, 2007.

[GH05] Tobias Gedell and Reiner Hähnle. Automating verification of loops by parallelization. Technical report, Department of Computing Science, Chalmers University of Technology, Department of Computing Science, Chalmers University of Technology S-412 96 Göteborg, Sweden, gedell,reiner@chalmers.se, 2005.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. The Java series. Addison-Wesley, Reading, MA, USA, third edition, 2005.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[IV02] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types, 2002.

[JMPS05] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In *7th Workshop on Formal Techniques for Java-like Programs - FTfJP'2005*, July 2005.

[Mal06] Ovidio José Mallo. MultiJava, JML, and Generics. Semester Project Report – ETH Zürich, 2006.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[MPMU03] C. Marche, C. Paulin-Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml, 2003.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[rec] Recoder – a java framework for source code metaprogramming. http://recoder.sourceforge.net.

[SC06] James Sasitorn and Robert Cartwright. Efficient first-class generics on stock java virtual machines. In *SAC*, pages 1621–1628, 2006.

[Ste05] Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, 2005.

[TEH05] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild fj. Presented at Foundations of Object-Oriented Languages (FOOL 2005), January 2005.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahe, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296, New York, NY, USA, 2004. ACM Press.

[vOP98] 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, 1998. http://isabelle.in.tum.de/Bali/papers/JavaDays98.html.

[Wid06] Florian Widmann. Crossverification of while loop semantics. Master's thesis, University of Karlsruhe, 2006.