Formal Correctness Proof of a Program Logic Calculus for the Deductive Verification of JAVA Programs

Studienarbeit

Bettina Sasse

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Dr. Bernhard Beckert

Universität Karlruhe (TH)
Fakulät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Contents

Chapter 1

Introduction

1.1 Motivation

The invention of SmartCards opened a new area of programs well suited for verification, as they are very small due to the limited memory capacities of the SmartCard chip. These programs usually require a high security standard as they carry sensitive data like personal data and medical information, bank account information, etc. In addition, the number of smart cards with the same program makes a replacement of the cards, because of software bugs very expensive. Verifying the software for SmartCards could assure that the stored data cannot be accessed by unauthorised persons or machines and guarantee 'bug-free' software.

In order to prove a program in a certain programming language a calculus which describes this language is needed. Valid proofs of programs can only be build with a correct calculus.

This paper describes an attempt to verify the rules of a calculus for JAVA CARD in Dynamic Logic using a formalisation of Java^{light}, another subset of JAVA, and the theorem prover Isabelle.

In order to prove the rules of the calculus for JAVA CARD in Isabelle, every rule needs to be expressed in a syntax and a type system that is understood by Isabelle. As we try to omit the construction of a formalisation of JAVA CARD suitable for Isabelle, we use an existing formalisation called Bali instead, which works on a slightly different subset of JAVA called Java^{light}. Having the Bali syntax as input for Isabelle, the rules of our calculus need to be expressed in this syntax. The results of the syntax adjustments can then be used as proof obligations and Isabelle as a proof aid to prove the rules correct.

1.2 Overview

The current chapter explains the basics about the calculi for JAVA CARD and Java^{light}. The explanations about the JAVA CARD calculus, called KeY calculus in this paper, start with the KeY project, which was the reason for developing the calculus. The next two sections discuss the differences between JAVA CARD and Java^{light} and cover the essentials of Dynamic Logic. A short introduction into David von Oheimbs formalisation of Java^{light}and into the proving system Isabelle/HOL finish the chapter.

1.3 The KeY project

The KeY project [1] aims to increase the use of proving mechanisms to enhance the quality of security critical programs. Knowing that most software engineers are non-professionals when it comes to formal methods the greatest challenge is to simplify the necessary tasks as much as possible.

Assuming that most software engineers already use a modelling language like UML or OCL which are integrated in most CASE tools we use these already existing UML and OCL specifications of programs to generate proof obligations. A proof would then, for example, compare the implemented program with its formal specification.

Without a tight connection of all proving steps neither a simple handling nor the acceptance of the users can be achieved. Hence the developing environment, in KeYTogetherJ [?] is used, needs to be extended with the proving system.

Dynamic Logic was chosen to suit the needs of the target group. In this logic it is not necessary to transform the JAVA CARD programs into logical formulas, they can directly be inserted into the formulas. This leads to an quite intuitive understanding of the formulas, which appear no longer as cryptic symbols but as programs with a preceding logic part, the precondition and a following logic part, the postcondition.

1.4 Java Card vs Java^{light}

JAVA CARD is a subset of JAVA which is adjusted to the needs of SmartCards. It contains neither garbage collection nor multithreating nor dynamic loading of classes. Java^{light} is a subset of JAVA CARD which serves as an example language to explain the way of formalizing the main language features, and therefore has no industrial applications.

In comparison to JAVA CARD, Java^{light} is further restricted as there are no `throws` clauses in method heads, no unary or binary operators, stack overflow or memory overflow when allocating a class object are not included. Java^{light} does not contain some features of JAVA CARD like nested blocks, `break`, `continue`, and `return` statements or instance fields, but those could be emulated.

According to these differences the transformation of the JAVA CARD calculus into the formal syntax and semantics defined for Java^{light} cannot be complete. Hence only a part of the rules of the KeY calculus is regarded here.

1.5 Dynamic Logic

Dynamic Logic (DL) is a modal predicate logic with two modal operators. Formulas without these operators are similar to formulas in first order predicate logic.

The extension of DL [3] we use, has instead of one type of variables two types, program and logical variables. Program variables are denoted by `x, y, z`. Their value may be changed by programs and though can differ in different states. They cannot be quantified or instantiated with terms. In contrast to them logical variables x, y, z have the same value in all states, have to be bound by a quantifier and can be instantiated with terms.

For each type of JAVA CARD exists a corresponding type in the JAVA CARD DL.

Logical terms are constructed from program variables, logical variables, constants and functions symbols of all types. There is a special postfix operator which can be applied to all terms of type pointer, so that the pointer is dereferenced and the object it points to is accessed.

Programs, that is JAVA CARD programs used in DL formulas, may contain logical terms instead of expressions if both have the same result type.

Formulas are built from logical terms, predicate symbols of all types, the logical equality operator $\doteq$, logical operators like $\neg$, $\wedge$, $\vee$, $\rightarrow$ and the quantifiers $\forall$ and $\exists$. In addition to this formulas may contain programs and two modal operators. Their semantics rely on the notion of a state.

A *state* does not contain information about the control flow like a program counter or a thrown exception. It contains the values of all program variables, attribute fields of objects, class attributes (static fields) of all classes in the context and the special reference `this`. According to this state definition, a program is equal to a number of state transitions, because it assigns to each initial state a set of possible final states.

The *modal operators* are connected to the notion of states. The diamond operator $\langle \mathrm{p} \rangle$ refers to all final states of program p which can be reached after the execution of p. As JAVA CARD is a deterministic language this is either one state or none depending on whether the program terminates or not. In contrast to this the box operator $[p]$ expresses the definite termination of program p and therefore only refers to one specific state.

Using one of the operators a formula like $\langle p \rangle \phi$ can be build. It is satisfied by a state s if the program is executed with initial state s and the formula ϕ in the final state s'.

An extension of this formula, $\psi \rightarrow \langle p \rangle \phi$, is valid for every state s which satisfies ψ if ϕ holds in the final state s' reached by running program p with initial state s.

$\psi \rightarrow \langle p \rangle \phi$ is equal to the Hoare triple $\{\psi\}p\{\phi\}$ except that ψ and ϕ are not allowed to contain programs in the Hoare triple.

1.6 Bali - A Formalization of Java

Formalizing a subset of JAVA in a way that allows to deduce properties of the programming language was the topic of David von Oheimbs PhD thesis [12]. His work is part of a project called Bali. He embedded Java^{light} , a subset of JAVA , in the theorem proving system Isabelle, developing a calculus, which will be called the Bali calculus in this paper.

David von Oheimb first introduced an abstract syntax for Java^{light} and then formalized a static semantics including JAVA's type system and wellformedness conditions. Next he designed an operational semantics, proved the soundness of the system and showed that the type safety claimed by JAVA in fact holds. As a last step he defines an axiomatic semantics of partial correctness based on Hoare logic, proves soundness and completeness of his axiomatic semantics with respect to the operational semantics.

Here we will only use his operational semantics. The next chapter explains in detail how he translated JAVA names, values, types etc into Isabelle types.

1.7 Isabelle

Isabelle [**?**][**?**] is a complex theorem prover which uses higher order logic (HOL) as standard logic. It was developed by Nipkow and Paulson and enhancements are still continuing. Prooving in Isabelle is semiautomatical. The user mainly interacts with the system to start important steps on a high abstraction level like case distinction or induction. In addition to that the user provides lemmas about properties of the involved entities, which are essential for the proof. These lemmas can be used automatically in the basic tactics like rewriting and resolution.

The general proof style is goal directed, backward chained, which means the goal is reduced to subgoals until the proof is trivial. The used syntax is close to mathematical notations and to the implementation language ML.

One of Isabelle's features is to deduce the type of a variable depending on the place where it appears. This is a significant help as the explicit definition of variables is no longer necessary. During this procedure Isabelle checks as well if the usage of a variable is type consistent, that is for every occurrence of a variable the same type has to be computed, and reports an error if this is not the case.

HOL (higher order logic) provides the basic types *unit*, *bool*, *int* and *nat*. More complex types can be combined using $(\alpha)set$ which produces homogeneous sets for any element type α. Product types are defined via $\alpha \times \beta$ which comes with the selector functions fst and snd. Pairs in tuples are always considered to be nested to the right, so $(a, b, c) = (a, (b, c))$.

A sum type $\alpha + \beta$ exists to cover a choice between types, which can be accessed via Inl and Inr. Finally a list type (α)list is available with constructor [].

A frequently used type is $(\alpha)option$ with the constructors None and Some α. It comes with an unpacking function the such that (Some x) $= x$ and None $=$ arbitrary. arbitrary is the default value.

Further datatypes can be defined like $nt = Ct_1t_2$ — ..., where nt is the new type and C is a constructor in the shape of a injective function $C :: t_1 \to t_2 \to nt$.

To enhance the readability type abbreviations can be introduced as equations. The declaration of logical constants follows the structure *name* :: *type*. For non recursive definitions and syntactic abbreviations $\equiv$ is used. Terms are expressed in an extended lambda calculus similar to the one used in ML.

Formulas appear in a standard way, the mathematical symbols have ASCII translations. To enhance readability, the mathematical symbols are used here. A conversion table can be found in the appendix.

Chapter 2

The Bali calculus

The Bali calculus was developed as a calculus for Java^{light}, a subset of JAVA, at the Technical University of Munich by David von Oheimb as part of his PhD thesis [12]. As we use his implementation of the calculus in Isabelle/HOL, the calculus consists mainly of a number of types like all formalizations in Isabelle.

This chapter explains syntax, semantics and the relation between the types of the Bali calculus. In the meantime a newer version has been released together with a new version of Isabelle, Isabelle2002. However to be able to run the proofs on Isabelle2002, it was necessary to adjust the proof file (see appendix A.4). Therefore anything occuring in the source code of the appendix which was not mentioned in this chapter is most likely due to change to the new version. The explanation below refers to the Bali version described in the PhD thesis.

To enhance the readability different fonts are used. JAVA names like `object` are written in typewriter. Type names of Isabelle types like *emane* and variables are displayed in italics and Isabelle constructors are written in slanted font.

Using the implementation of the calculus requires to pay attention to non-logical subjects as well. For example the number of white spaces used sometimes matters. When definitions are used, it is important that exactly the same pattern as in the place where the definition is defined is used.

2.1 Isabelle types for Java Names

Names of JAVA objects are grouped into three categories of types in the Bali calculus: *mname* is the Isabelle type for JAVA method names, *ename* stands for expression names and covers field and local variable names and *tname* for type names such as user defined types, exception names, class and instance names.

$$
\begin{array}{rcl}
tname & = & \texttt{Object} \\
& | & \mathsf{SXcpt}\ xname \\
& | & \mathsf{TName}\ tnam \\
\\
xname & = & \texttt{Throwable} \\
& | & \texttt{Nullpointer} \\
& | & \texttt{OutOfMemory} \\
& | & \texttt{ClassCast} \\
& | & \texttt{NegArrSize} \\
& | & \texttt{IndOutBound} \\
& | & \texttt{ArrStore}
\end{array}
$$

tnam is the type for class and instance names, while SXcpt (standard exception) is the constructor for the in this Bali version specified seven exceptions names. `Object` is the name of the default JAVA class.

In order to group local variables and the `this` pointer which is not included in the upper list, a sum type *lname* is introduced:

$$lname = \text{"}ename + unit\text{"}$$

To access both parts of the sum type, the abbreviations of the required functions are named EName and This.

2.2 Primitive and Reference Types of Java

The most general distinction of types is the division into primitive and reference types:

$$\begin{array}{lcl} ty & = & \mathsf{PrimT}\ prim\_ty \\ & | & \mathsf{RefT}\ ref\_ty \end{array}$$

Primitive types are further divided into the `void` type, boolean values and integers. Other primitive JAVA types like `double` or `short` are not covered in this version of the calculus.

$$\begin{array}{lcl} prim\_ty & = & \texttt{void} \\ & | & \texttt{boolean} \\ & | & \texttt{int} \end{array}$$

Reference types are divided into interface, class and array types. For sake of simplicity, the `null` type is considered to be a reference type as well.

$$\begin{array}{lcl} ref\_ty & = & \mathsf{IfaceT}\ tname \\ & | & \mathsf{ClassT}\ tname \\ & | & \mathsf{ArrayT}\ tname \\ & | & \mathsf{NullT} \end{array}$$

In order to refer to the reference types directly by their names the following abbreviations are introduced:

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

2.3 Java Values

The values attached to the types follow the standard HOL values in case of *bool* and *int*. Unit is a special dummy value used for the result of statements and methods with return type `void`. The type *loc*, representing the value of non-null references to objects is not further specified.

$$\begin{array}{lcl} val & = & \mathsf{Unit} \\ & | & \mathsf{Bool}\ bool \\ & | & \mathsf{Intg}\ int \\ & | & \mathsf{Null} \\ & | & \mathsf{Addr}\ loc \end{array}$$

In order to retrieve the type of a value, a function typeof is specified which has an optional output, because addresses can only return a type if they point to an object.

Every type has a default value, which is set according to the JAVA language specification [5].

2.4 Java Terms

JAVA terms are divided into statements, expressions and variables. Note that not every JAVA term is formalized because Java^{light} does not cover the whole range of JAVA terms. If an abbreviation for a constructor exists, it is listed here instead of the constructor.

var	=	LVar $lname$	local variable
	|	$\{tname,\ bool\}expr..\ ename$	class field
	|	$expr.[expr]$	array component

Local variables are defined with an explicit constructor to avoid ambiguity. The argument *lname* can either be the name of a local variable preceded by constructor Ename, or This which points to the current object.

A class field is built with a rather complex structure. The last argument *ename* is the name of the class field. The expression describes a class C which is the starting point for a search in the class hierarchies for a class with a class field with the specified name. The first parameter in the type annotations in braces gives the first match of the search followed by a boolean value stating if the class field is defined static or not.

The array component consists of an expression which specifies the array and a second expression in brackets which selects a certain component of the array.

$expr$	=	NewC $tname$	class instance creation
	|	New $ty[expr]$	array creation
	|	Cast $ty\ expr$	type cast
	|	$expr$ InstOf ref_ty	instanceof
	|	Lit val	literal value
	|	Super	keyword `super`
	|	Acc var	variable access
	|	$var := expr$	variable assign
	|	$expr?expr : expr$	conditional
	|	$\{ref\_ty, ref\_ty, inv\_mode\}expr..mname$	
		$(\{(ty)list\}(expr)list)$	method call
	|	Methd $tname\ sig$	(folded) method
	|	Body $tname\ stmt\ expr$	(unfolded) method body

The class instance creation expression consists of a constructor and the class name of the class to be instantiated. The creation of an array has two parameters: *ty* states the type of the array components, for example `bool` or `int`, and the expression in brackets specifies the size of the array. A cast expression transforms the result of an expression into the given type *ty*. The InstOf operator is the Bali equivalent to the JAVA `instanceof` operator. Literal values are preceded with the constructor Lit and variable access is indicated by Acc. The JAVA keyword `super` is written analogous to `this` as Super. It contains the same value as This but its type is the super type of This. Apart from a different assignment symbol, the syntax of a variable assignment is similar to the JAVA assignment.

The tertiary operator ?: remains completely unchanged, in contrast to the method declaration which includes various additional information in the Bali version. The method name is of type *mname*. The expression specifies a class, which is the starting point for a look up procedure when the method is called. The result type of the expression is given as the first argument of the first annotation. The class which actually defines the desired method - it is either the class given by the expression or a superclass of it - is kept as the second *ref_ty* of first annotation. *inv_mode* states the invocation mode of a method call which can be either Static, Super or IntVir, which leaves the choice between `interface` and `virtual`. This parameter helps to handle all kinds of method calls in a single rule. The type list of the second annotations specifies the types of the method parameters which are given in the following expression list.

The signature of the method expression consists of the method name and a type list of its parameters. This signature is unique for all methods of a class. Accompanied with the name of the class, it addresses - used with the constructor Methd - the folded implementation of the called method. The unfolded version is represented by the Body expression, which includes the defining class, the statements of the body and the return expression.

$$
\begin{array}{lcll}
\mathit{stmt} & = & \mathsf{Skip} & \text{empty statement} \\
 & | & \mathsf{Expr}\ \mathit{expr} & \text{expression statement} \\
 & | & \mathit{stmt};;\ \mathit{stmt} & \text{composition} \\
 & | & \mathsf{If}(\mathit{expr})\ \mathit{stmt}\ \mathsf{Else}\ \mathit{stmt} & \text{if-then-else} \\
 & | & \mathsf{While}(\mathit{expr})\ \mathit{stmt} & \text{while loop} \\
 & | & \mathsf{Throw}\ \mathit{expr} & \text{throw} \\
 & | & \mathsf{Try}\ \mathit{stmt}\ \mathsf{Catch}(\mathit{tname}\ \mathit{ename})\ \mathit{stmt} & \text{try-catch statement} \\
 & | & \mathit{stmt}\ \mathsf{Finally}\ \mathit{stmt} & \text{finally statement} \\
 & | & \mathsf{init}\ \mathit{tname} & \text{class initialization}
\end{array}
$$

Note that in difference to JAVA, none of the statements formalized in Bali is finished with a semicolon. Here, a semicolon is used only for the separation of statements and for this purpose in a double form. This design makes it necessary to change the empty statement in JAVA ";" into Skip. In order to use an expression instead of a statement, the constructor Expr is added to the expression.

The syntax of the If statement is similar to the JAVA version except for the capital letters. The while and throw expression follow the same principle. The JAVA `try-catch-finally` statement is divided into two separate statements and may contain only one catch clause. Several catch clauses can be simulated using the InstOf operator. The init statement is inserted into all parts of the program, at places where a class might be initialized. It does not exits in the original language. When called, it tests if the class given in its argument has already been instantiated. If so nothing is done. Otherwise the class is initialized and all of its super classes as well, in case they have not been initialized yet.

Notice that no block is defined, so it is not valid to use braces in combination with any of the statements.

2.5 Fields and Methods of Java

$$
\begin{array}{lcll}
\mathit{modi} & = & \mathit{bool} & \text{modifier} \\
\mathit{field} & = & \mathit{modi} \times \mathit{ty} & \text{field} \\
\mathit{fdecl} & = & \mathit{ename} \times \mathit{field} & \text{field declaration}
\end{array}
$$

Field and method modifiers are currently restricted to `static`, which makes it possible to model it as a boolean type. The field declaration consists of a name *ename* and the field itself, which is built from a modifier *modi* and a type *ty*.

$$
\begin{array}{lcll}
\mathit{mdecl} & = & \mathit{sig} \times \mathit{methd} & \text{method declaration} \\
\mathit{methd} & = & \mathit{mhead} \times \mathit{mbody} & \text{method} \\
\mathit{mhead} & = & \mathit{modi} \times (\mathit{ename})\mathit{list} \times \mathit{ty} & \text{method head} \\
\mathit{mbody} & = & (\mathit{ename} \times \mathit{ty})\mathit{list} \times \mathit{stmt} \times \mathit{expr} & \text{method body}
\end{array}
$$

A method consists of its signature, which is unique for every method of a class, and of the method itself, which can be split into method head *mhead* and method body *mbody*. The method head includes a modifier which states whether the method is declared `static` or not, the parameter list *ename(list)* and the return type *ty*. The method body is build from a list of local variables, a statement which represents the body, and the result expression. This expression is a replacement for the `return` statement, and has the advantage, that `return` statements no longer can appear in arbitrary places in a method body. `throws` clauses at the method head are not covered in the Bali calculus.

2.6 Interfaces and Classes

$$\begin{array}{lcll} idecl & = & tname \times iface & \text{interface declaration} \\ iface & = & (tname)list \times ibody & \text{interface} \\ ibody & = & (sig \times mhead)list & \text{interface body} \end{array}$$

Interface declarations are specified by the name *tname* of the interface and the interface itself. The interface consists of a list of super interfaces *(tname)list* and an interface body. The body contains nothing but a list of method declarations *(sig* $\times$ *mhead)list*.

$$\begin{array}{lcll} cdecl & = & tname \times class & \text{class declaration} \\ class & = & tname \times (tname)list \times cbody & \text{class} \\ cbody & = & (fdecl)list \times (mdecl)list \times stmt & \text{class body} \end{array}$$

Classes are build following the same scheme as interfaces. Additionally they refer to the name of the superclass with the first type *tname* in type *class*. The class body consists of a list of field declarations, a list of method declarations and a statement which contains the method body.

2.7 Java Programs

$$prog \quad = \quad (idecl)list \times (cdecl)list \quad \text{program}$$

is the definition for a program. Two functions simplify the access to a class and an interface: iface G I and class G C. G is used in this paper for a program in the Bali calculus (derived from the originally used Γ in order to avoid confusion with the antecedent Γ in DL formulas), I and C stand for an interface and a class.

2.8 Stores

Two stores keep track of the global objects and locals variables. Both are kept in a table, with the object reference *oref* and local variable name *lname* as key to access the stored object or value.

$$\begin{array}{lcll} globs & = & (oref, obj)table & \text{global variable} \\ locals & = & (lname, val)table & \text{local variable} \end{array}$$

An object reference *oref* is either the address *loc* of the object on the heap or the class name *tname* of an object. The object itself is a pair of a tag *obj_tag* and a table which holds its fields and values. This table can be accessed by variable names *vn*, which are either field specifications *fspec*, consisting of the field name *ename* and the name of the defining class *tname*, or by an integer in case of access to an indexed value of an array.

$$\begin{array}{lcll} oref & = & loc + tname & \text{object reference} \\ obj & = & obj\_tag \times (vn, val)table & \text{object} \\ obj\_tag & = & \mathsf{CInst}\ tname & \text{object tag} \\ & | & \mathsf{Arr}\ ty\ int & \\ vn & = & fspec + int & \text{variable name} \\ fspec & = & ename \times tname & \text{field specification} \end{array}$$

The two tables for global objects and local variables are then combined in the store type *st*. For retrieving one of the variables, two access functions globs and locals are defined.

$$st \quad = \quad \mathsf{st}\ globs\ locals \quad \text{store}$$

2.9 State

Apart from objects and values it is also necessary to keep track of the exceptions, so the store type is extended by *xopt* to the state type *state*.

$$\begin{array}{lcll} state & = & xopt \times st & \text{state} \\ xopt & = & xcpt\ option & \text{optional exception} \\ xcpt & = & \mathsf{XcptLoc}\ loc & \text{exception} \\ & | & \mathsf{StdXcpt}\ xname & \end{array}$$

Exceptions can either be included in a state using the address of the allocated exception object as a reference or using the name of an standard exception as reference. As not every state contains an exception, they are made optional and an abbreviation for states without exceptions is introduced:

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

2.10 Judgments

Now all datatypes are defined and can be combined to form a judgment. Here is the general scheme:

$$prog \vdash state_{init} - term \succ\!\rightarrow (val \times state_{final}).$$

prog is the program in whose context *term* is executed starting in state $state_{init}$ reaching state $state_{final}$ and returning value *val*. To simplify the use of the judgments, several abbreviations are available, depending whether *term* is a statement, expression, variable or an expression list:

$$\begin{array}{r} prog \vdash state_{init} - stmt \ \rightarrow\ state_{final} \\ prog \vdash state_{init} - expr \multimap\!\succ val \rightarrow state_{final} \\ prog \vdash state_{init} - var \ \Rightarrow\!\succ vval \rightarrow state_{final} \\ prog \vdash state_{init} - (expr)list \ \dot{\Rightarrow}\!\succ (val)list \rightarrow state_{final} \end{array}$$

2.11 Exception Handling in Bali

If an exception is thrown, the computation of all subsequent statements is skipped and the exception is propagated to enclosing statements until it is caught or the program execution terminates. As exceptions can occur nearly everywhere, the final state of most judgments may contain an exception. If a rule causes a split from one into two judgments, the rule usually has to be split in two rules: one for the case that the intermediate state (the state reached after the first judgment) contains no exceptions and the other one to describe the case that an exception occurred in the first judgment. This rule split becomes very complicated and large as soon as more than two sub-terms are involved. To avoid this, the Bali calculus allows exceptions in intermediate states and handles the execution of a term with an exception in the initial state with the following rule:

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$$

$G \vdash (\mathsf{Some\ xc}, \mathsf{s}) - t \succ \mathsf{arbitrary3}\ t \to (\mathsf{Some\ xc}, \mathsf{s})$1em] The function arbitrary3 provides an arbitrary return value, which is either arbitrary or Unit depending on the type of term t. In case an exception is present, the return value of a term is irrelevant, but the inclusion of it unifies the appearance of the rule.

As the current version of the Bali calculus does not contain `break` or `continue` statements, the handling of abnormal statements is limited to the `throw` statement. The `throw` rule first evaluates the reference of the thrown exception x. If no exception occurs during this evaluation and if the reference is not a null pointer, an auxiliary function throw copies the evaluated location a' into the exception component of the final state:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - e \succ a' \to s_1}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{throw}\ e \to \mathsf{xupd}\ (\mathsf{throw}\ a')\ s_1}$$

1em] Exceptions in this version of the calculus, can only be caught if they occur in a `try` statement because a `throws` appendix for a method head is not modeled.

$$\frac{\begin{array}{c} G \vdash \mathsf{Norm}\ s_0 - c1 \to s_1 \quad G \vdash s_1 - \mathsf{sxalloc} \to s_2 \\ \mathsf{if}\ (\Gamma, s_2 \vdash \mathsf{catch}\ C)\ \mathsf{then}\ (G \vdash \mathsf{new\_xcpt\_var}\ vn\ s_2 - c2 \to s_2)\ \mathsf{else}\ s_3 = s_2 \end{array}}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{try}\ c1\ \mathsf{catch}(C\ vn)\ c2 \to s_3}$$

1em] The first judgment $G \vdash \mathsf{Norm}\ \mathsf{s}_0 - c1 \to s_1$ evaluates the `try` block. The resulting state s_1 may contain an exception. If this is the case and the exception is a standard exception which is represented by its name and not an address, the function sxalloc replaces the name of the exception with its address. In every other case sxalloc is the identity function of the state, that is the initial state is equal to the final state. The third formula of the premisses is an Isabelle if function. If an exception has been raised in the `try` block and if it is assignable to the `catch` clause (same class or subclass of the parameter of the `catch` clause), the exception is bound to the parameter new_xcpt_var vn s_2 and the `catch` clause is executed. The binding of the exception to a local variable makes the exception harmless, as it is no longer contained in a state. In every other case, that is no exception has been thrown in the `try` block or the thrown exception cannot be caught by the `catch block`, the final state s_3 is the same as s_2, the state after the second judgment.

2.12 Bali Rules

This part describes a few rules exemplary to show the mechanisms of rules in the Bali calculus. For a full list see [12]. In general the rules are to be applied - as well as read - from bottom to top.

One of the most simple rules handles the expression statement:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - e \succ v \to s_1}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ e \to s_1}$$

1em] An expression statement Expr e of program G is to be executed starting in an exception free state s_0, reaching an not further known state s_1. In order to calculate the value of the contained expression e, the constructor Expr has to be deleted. Doing so the statement becomes an expression, which has in contrast to a statement a return value. The return value of e is displayed as v.

The following rule is slightly more complicated, as it needs a split into two formulas to handle the composition of two statements.

$$\frac{G \vdash \mathsf{Norm}\ s_0 - c_1 \to s_1 \quad G \vdash s_1 - c_2 \to s_2}{G \vdash \mathsf{Norm}\ s_0 - c_1;;\ c_2 \to s_2}$$

1em] The rule transforms the composition of two statements c_1 and c_2 into two separate statements. The statements are part of program

G. The initial state s_0 does not contain any exceptions as indicated by Norm. Because we do not have any knowledge about c_1 and c_2 we cannot assume that no exceptions are contained in the final state s_2 or the intermediate state s_1, which maps the state of the program after the execution of c_1 and before the execution of c_2.

A third class of rules contains built-in Isabelle functions. An example for this is the If rule:

$$\frac{G \vdash \mathsf{Norm}\ s_0\ - e \succ\!\!\rightarrow b \rightarrow s_1 \quad G \vdash \mathsf{s_1} - \mathsf{if\ the\_Bool}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \rightarrow\ s_2}{G \vdash \mathsf{Norm}\ s_0\ - \mathsf{If}(e)\ c_1\ \mathsf{Else}\ c_2\ \rightarrow\ s_2}$$

1em] Starting from an exception free initial state s_0, an If statement is executed in the context of program G. This If statement is split into two separated formulas. The first one evaluates the value b of the condition e and reaches state s_1. The actual case distinction is done via a built-in if-then-else function of Isabelle/HOL in the second judgment. The Isabelle if function uses the previously computed value b as condition and the unchanged statements c_1 and c_2 as if and else block. As b is of type *val* but a boolean is required as condition, the function the_Bool is used to access the underlying boolean value.

The another type of rules contains built-in Isabelle functions which are not part of a judgment, but include one. An example is the While rule:

$$\frac{G \vdash \mathsf{Norm}\ s_0\ - e \succ\!\!\rightarrow b \rightarrow s_1 \quad \mathsf{if\ the\_Bool}\ b\ \mathsf{then}\ (G \vdash s_1 - c\ \rightarrow\ s_2\ \wedge\ (1))\ \mathsf{else}\ s_3 = s_1}{G \vdash \mathsf{Norm}\ s_0\ - \mathsf{While}(e)\ c\ \rightarrow\ s_3}$$

(1) $G \vdash s_2 - \mathsf{While}(e)c\ \rightarrow\ s_3$ 1em] A single application of this rule unwinds the While loop one time so that one iteration of the loop can be executed. Analogous to the If rule, the condition e is evaluated first. The second judgment tests using the built-in Isabelle if function, whether the value b of e is true or not. If the condition is true, the loop will be entered and the body c is executed once, reaching state s_2. Then the whole loop is possibly executed again, reaching state s_3. If b is found to be false, the loop is skipped instead of being executed, so that after the evaluation of e the state does not change anymore. Hence the final state s_3 is equal to s_1.

Chapter 3

The KeY calculus

The KeY calculus was defined for JAVA CARD programs. Only a single restriction applies to the programs: inner classes, i.e class definitions inside of classes, are not allowed. However, this does not decrease the range of programs, as inner classes can be transformed into normal classes so that all classes form a sequent. This sequent of classes is called a context.

Most operations in the calculus base on rewriting the JAVA programs. To be able to stick to this principle while accessing information about the current state of the program, it is assumed that for every class C several special fields and methods are implicitly defined[3]:

```
public static C firstObj; public static C lastCreatedObj; public C prevObj;
public C nextObj; public boolean beforeObj(C obj); public boolean created;
public static boolean classInitialised; public boolean objInitialised;
```

These fields and methods provide a list of all objects of the class by the pointers `firstObj`, `prevObj` and `nextObj`. Additional information about initialised objects includes a pointer `lastCreatedObj` to the newest created object, a boolean variable `created` stating whether an object has been created or not, another boolean value `classInitialized` which is true if the class has been initialised and a third one `objInitialized` providing information whether an object has been initialised. To keep these variables up to date, every time the `new` operator is used, it changes the value of the pointer `lastCreatedObj` of the instantiated class to the new object and sets the `created` flag of the new object to true. The method `beforeObj` returns true if the pointer to the current object `this` is in the list at a place before the argument of the method `obj`.

3.1 Variables

To avoid confusion, the calculus distinguishes between JAVA program variables, displayed in `typewriter` and logical variables displayed in *italics*. The value of program variables may change from state to state, depending on the statements of the program. They cannot be quantified or instantiated with terms. Logical variables act differently. They must be quantified and can be instantiated with terms but must not change their value when the state changes.

3.2 Types

The set of types used in the calculus contains each primitive JAVA CARD type such as `int` or `byte`, each built-in class like `Object` or `String` and each class which is defined in the context. Furthermore a pointer type, an array type and a pointer to an array type exist.

3.3 Terms

Logical terms are constructed from both program and logical variables, string literals, all Java Card literals for the primitive types and the `null` reference. Constructions like `o.a` are a term as well, where `o` is a term denoting an object of a class and `a` is a field of that object. The same construction is a valid term if `o` is a class name and `a` is a static field of that class. For arrays terms like `a[i]` can be produced, where `a` is an array type term and i is a term of type `int`.

3.4 State

In every state the values of program variables, fields of an object, static fields of all classes in the context and the pointer to the current object `this` can change their value compared to a different state. Variables of a reference type point to an object of the appropriate type, which can be replaced by a different object of the same type in every state. States do not contain information about the control flow or exceptions. This information is explicitely included as statements in the Java program.

3.5 Programs

Initially every program in a DL formula is executable code. However, after the application of some special rules this may change. Valid programs may contain assignments, method calls, unary and binary operators, the tertiary operator `?:`, blocks with or without labels, `if-else`, `switch` and `try-catch-finally` statements, `while`, `do-while` and `for` statements. There are `throw`, `break` and `continue` statements as well as `return` statements which end a method call. As it is necessary to remember the variable to which the computed value should be assigned when a method call returns, a special syntax for unfolded method calls is needed. It is designed like a labelled block with `methodcall(`*old*`,` x`)` as label. *old* is used to store the value of the `this` pointer at the point when the method was called and x is the variable to which the returned value is assigned. If the return value is not assigned to a variable, the `methodcall` label contains only the single parameter *old*. Inside the block the appropriate method body is displayed, preceded with a list of assignments of the parameter values of the method call to the parameter variables.

The semantics of a program is closely tight to state transitions. A program p started in state s assigns to s a set of states which can be reached by one run of p. As Java Card is deterministic, such a set contains either one state or is empty in case the program does not terminate. A logical term which occurs in a program has the same properties and value as an expression without side-effects. Programs which terminate not normally, e.g with an exception, are regarded as non-terminating.

3.6 Formulas

Formulas are built from logical terms, predicate symbols of all the types and logical connectives like $\vee$, $\wedge$, $\neg$ and $\rightarrow$. The quantifiers $\forall$ and $\exists$ can only be applied to logical variables and not to program variables. Formulas may include the logical equality operator $\doteq$ and the modal operators $[\,]$ and $\langle\rangle$, that is if p is a program and ϕ is a formula, then $[p]\phi$ and $\langle p\rangle\phi$ are formulas, too.

If a variable o of a certain class type is quantified in a formula $\forall o \phi(o)$, the quantification ranges over the set of all objects of this class type, no matter whether they have been created or not.

Formulas without programs have the same semantics as they would in first-order logic. A formula $\langle p\rangle\phi$ is satisfied by a state s, if p started in s terminates in s' and ϕ is satisfied in s'. A formula is satisfied by a model, if there is a state in the model which satisfies the formula. A formula is valid in a model, if it is satisfied by all states of the model. A formula is valid, if it is valid in all possible models.

3.7 Updates

Updates of the form $\{x \leftarrow t\}$ or $\{$o.a $\leftarrow t\}$ are used if the value of a program variable x or a field a of an object o changes. They can be attached to terms and formulas. The semantics of an update is, that the formula or term it is attached to is to be evaluated in an "updated" state which is reached by replacing the variable or field with term t. Another way to write $\phi^{\{x \leftarrow t\}}$ is $\langle x = t; \rangle \phi$.

3.8 Sequents

A sequent is of the form $\Gamma_1 \ldots \Gamma_n \vdash \Delta_m \ldots \Delta_p$, with Γ_i $(i = 1 \ldots n)$, Δ_j $(j = m \ldots p)$ being formulas. It has the same semantics as $\bigwedge_{i=1}^{n} \Gamma_i \rightarrow \bigvee_{j=m}^{p} \Delta_j$

3.9 Exception handling

The JAVA language specification [5] categorises reasons for abrupt completion as followed:

- `return` statements without value
- `return` statements with return value
- `break` statements without label
- `break` statements with label
- `continue` statements without label
- `continue` statements with label
- `throw` statements

These statements cause an immediate interruption of their enclosing statement. These enclosing statements then terminate either normally or abruptly, depending on the combination of reason and statement. A `return` statement for example ends the execution of a method body immediately, but this termination of the method is considered to be normal, as a `return` is a valid reason to terminate a method call. Therefore the execution of the program will continue normally. Assume the `return` statement is enclosed in a loop, which is part of a method body. The loop will terminate abruptly, as loops are not meant to be completed by `return` statements. Terminating abruptly means, the reason for abrupt completion - a `return` statement in this case, does not disappear but stays in the program, where it is treated in combination with the surrounding method block.

As shown in this example, reasons for abrupt completion including exceptions are always handled inside the program. This means the reason for abrupt completion stays in the program and terminates enclosing statements abruptly until an enclosing statement is reached, which matches the reason and terminates normally. If no enclosing statement of this kind exists, the program is said to terminate abruptly, which is considered to be the same as non termination of a program.

3.10 Rules

KeY rules are of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

1em] where $\Gamma_i \vdash \Delta_i$ are sequents. $\Gamma \vdash \Delta$ is called the conclusion, $\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n$ are the premises. In a sequent $\Gamma \vdash \Delta$ Γ is called the antecedent and Δ is the succedent.

A proof is started with a formula which contains the complete program. Rules are applied in a bottom-up style, until an empty program is reached. The remaining logic part is solved in first-order logic. The appropriate rule to apply is chosen by the first executable statement. In general, programs are presented as $\pi s \omega$, where s is the first executable statement and π contains labels, opening braces "{", beginnings of method invocations "`methodcall( ... ):{`" and beginnings "`try{`" of `try-catch-finally` statements. ω contains the rest of the program, in particular the matching endings of π.

One of the most simple rules is the rule which handles the tertiary operator `?:`. This rule performs a transformation of the conditional statement into an equal `if-else` statement.

$$\frac{\Gamma \vdash \langle \pi \ \texttt{if(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ cnd \ \texttt{?}a\texttt{:}b \ \omega \rangle \phi, \Delta}$$

1em] Γ contains all information about the program in diamonds which has been collected so far, it is the precondition for a further execution of it. As no variable changes its value in the transformation step neither an update nor a change of the antecedent is needed.

For some statements two or more different rules exist, as they are designed to perform small steps, so that the program does not look completely different after a rule application. One of the statements with several rules is the `if-else` statement. The first rule to apply is usually the *if-condition-right* rule:

$$\frac{\Gamma \vdash \langle \pi \ \texttt{bool} \ b \ \texttt{=} \ cnd\texttt{;} \ \texttt{if(}b\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ \texttt{if(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}$$

1em] This rule is only applied when the condition *cnd* is not a simple expression, that is either a variable name or one of the values true or false. The rule introduces a new variable b of type `bool` to which the value of *cnd* is assigned. Then the new variable is used as `if` condition, so that the `if` statement is now has a simple condition. Statements with a simple expression as condition can then be treated with the main `if` rule, named *if-right*:

$$\frac{\Gamma, cnd \doteq true \vdash \langle \pi \ \texttt{a} \ \omega \rangle \phi, \Delta \quad \Gamma, cnd \doteq false \vdash \langle \pi \ \texttt{b} \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ \texttt{if(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}$$

1em] In this rule the `if` statement itself is executed. It is split into two parts, one covering the case that the condition is true, the other one evaluating the program under the assumption that the condition has the value false ($cnd \doteq false$). The next evaluation step then possibly eliminates one of the cases. In this case, an abbreviation can be used:

$$\frac{\Gamma \vdash cnd = true \quad \Gamma \vdash \langle \pi \ \texttt{a} \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ \texttt{if(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}$$

1em] The first premiss requires to draw the conclusion from Γ that the value of *cnd* is true. The second premiss then replaces the `if` statement with the content of the `if` block a, as this is the block to be executed. A similar rule exists for the case that the condition is false.

There are only a few rules which change the value of variables and therefore need updates. One of them is the assignment rule.

$$\frac{\Gamma^{\{x\leftarrow c\}}, x \doteq expr^{\{x\leftarrow c\}} \;\vdash\; \langle\pi\;\; \omega\rangle\phi^{\{x\leftarrow c\}}, \Delta^{\{x\leftarrow c\}}}{\Gamma \;\vdash\; \langle\pi\;\; x\; \texttt{=}\; expr\texttt{;}\;\; \omega\rangle\phi, \Delta}$$

1em] The assignment rule can only be applied if *expr* is a logical term. Otherwise *expr* needs to be evaluated first. In order to assign x the value of *expr*, a new constant c is introduced. c is assigned to every x, so that now x with value *expr* is shown as c. A direct replacement as in classical DL is not possible because of the reference types. Assume that l and m point to the same object. Now the object is exchanged using l as access pointer. If then l is replaced by n, m would still point to the old object and not to the new one. This problem is solved by not including updates into the program, but by using special rules for the simplification of updates.

One of the rules which involves a lot of changes in parts of the program is the *while-right* rule.

$$\frac{\Gamma \;\vdash\; \langle\pi\;\; \texttt{if(}a\texttt{)}\;\; l'\texttt{:\{}l''\texttt{:\{}\;\; b'\texttt{\}}\;\; \lambda\texttt{while(}a\texttt{)\{}b\texttt{\}\}}\omega\rangle\phi, \Delta}{\Gamma \;\vdash\; \langle\pi\;\; \lambda\texttt{while(}a\texttt{)\{}b\texttt{\}}\;\; \omega\rangle\phi, \Delta}$$

1em] The general structure of this rule unwinds the while loop one time and places it in an `if` statement to evaluate the condition. As usual, π and ω contain the beginning and end of the program. λ is an abbreviation for a sequence of labels, which may be empty. As the loop body `b` may contain `break` or `continue` statements with or without label, which reorganize the control flow, this behaviour has to be maintained in the if statement. First every `continue` is changed into a break as `continue` statements are only allowed in loops. Then the labels are adjusted. Two new labels are introduced, *l'* for `break` statements and *l"* for `continue` statements whose target is the current `while` loop. That means the labels of `break` or `continue` statements which have a label that is listed in λ are exchanged against *l'* or *l"*. In addition each break or continue without label gets the newly introduced labels. After this transformation, only `breaks` with either label *l'* or *l"* are left, or `break` and `continues` with a label that is not listed in λ. As the matching block for *l'* is the whole `if` block, the complete loop is ended if an original `break` occurs in *b'*. If a former `continue` is reached, the current loop iteration, which is the unfolded body *b'*, is completed immidiatedly and the execution of the loop continues with another test of the loop condition.

The rules for the `try-catch-finally` statements are divided into two sections: one for `try-catch-finally` statements without `finally` block and one for statements with `finally` block. If an exception occurs in the `try` block, the first matching catch block has to be found. This done by a mechanism of nested `if` statements, which uses a the `instanceof` operator in the `if` condition.

$$\frac{\Gamma \;\vdash\; \langle\pi\;\; \texttt{if(}cnd_1\texttt{)\{}c_1\texttt{\}else if(}cnd_2\texttt{)}\;\ldots\;\texttt{if(}cnd_m\texttt{)\{}c_n\texttt{\}}\;\texttt{else\{throw}\;x\texttt{;\}}\;\;\omega\rangle\phi, \Delta}{\Gamma \;\vdash\; \langle\pi\;\; \texttt{try\{throw}\;x\texttt{;}\;\; a\texttt{\}}\;\;\texttt{catch(}E_1\; exc_1\texttt{)}\;\;\texttt{\{}b_1\texttt{\}}\;\ldots\;\texttt{catch(}E_n\; exc_n\texttt{)\{}b_n\texttt{\}}\;\;\omega\rangle\phi, \Delta}$$

1em] $cnd_i \equiv x$ instanceof $E_i c_j \equiv exc_j = x$; b_j 1em] If a `throw` statement is reached inside the `try` block, the `catch` clauses are tested one after the other until one is found which has an argument of the same type as the thrown exception or the type of the thrown exception is a subclass of the type specified in the `catch` clause. This behaviour is modeled by nested `if` statements. If no matching `catch` clause exits, the `try` statement terminates abruptly with reason `throw` x. This behaviour is preserved in the nested *if* conditions, as the last else block contains the `throw` statement. This block will be chosen if none of the `if` conditions is true. The execution of the last `else` block causes the `if` statement to terminate abruptly with reason `throw` x, that is the `if` statement does not exist anymore, but the `throw` statement still exists, so the next surrounding statement in π has to handle it or is terminated abruptly by it.

There are rules to handle exceptions in a `try` statement without `catch` clauses and a rule which handles a `try` statement with an empty `try` block, for example because all statements of the block have been executed already.

Reasons for an abrupt completion (rfac) which do not match the surrounding type of statement are grouped together into one rule. An example is the *tc-try-rfac-right* rule which applies to all cases where a rfac exect a `throw` statement occurs as first executable statement in the `try` block.

$$\frac{\Gamma \vdash \langle \pi \;\; e;\;\; \omega\rangle\phi, \Delta}{\Gamma \vdash \langle \pi \;\; \texttt{try}\{e;\;\; a\}\;\; \texttt{catch}(E_1\;\; exc_1)\;\; \{b_1\}\;\; \ldots\;\; \texttt{catch}(E_n\;\; exc_n)\{b_n\}\;\; \omega\rangle\phi, \Delta}$$

1em] e is an abbreviation for `break;`, `break` l;, `continue;`, `continue` l;, `return` or `return` v;. Once one of these statements is reached, the `try` statement is aborted, that is deleted, and the enclosing statement, hidden in π and ω is deals with it if it is a matching statement or terminates itself. `Try` statements with a `finally` block are similar to `try` statements without `finally` block except for the treatment of the `finally` block. The rule to handle a `throw` in the `try` block is given as example:

$$\frac{\Gamma \vdash \langle \pi \;\; \texttt{try}\{\texttt{if(}cnd_1\texttt{)}\{c_1\}\texttt{else}\;\ldots\;\texttt{if(}cnd_m\texttt{)}\{c_n\}\;\;\texttt{else}\{\texttt{throw}\;\; x;\}\}\;\;\texttt{finally}\{f\}\;\;\omega\rangle\phi, \Delta}{\Gamma \vdash \langle \pi \;\; \texttt{try}\{\texttt{throw}\;\; x;\;\; a\}\;\; \texttt{catch(}e_1\texttt{)}\;\; \{b_1\}\;\; \ldots\;\; \texttt{catch(}e_n\texttt{)}\{b_n\}\;\; \texttt{finally}\{f\}\;\; \omega\rangle\phi, \Delta}$$ 1em]

$$\begin{array}{rcl} cnd_i & \equiv & x instanceof E_i \\ c_j & \equiv & exc_j = x; b_j \\ e_k & \equiv & E_k\, exc_k \end{array}$$

Here the `try` block has to be maintained, because in case an exception is thrown in the `catch` clause or `if` block, the `finally` block will be executed immediately. If an exception is thrown in the `finally` block, this exception will overwrite the former exception and the `try` statement terminates immediately abruptly because of the new exception. This behaviour is covered by the following rule, which is used if an exception is thrown in a `try` statement without `catch` clauses, but with a `finally` block.

$$\frac{\Gamma \vdash \langle \pi \;\; \{f;\;\; \texttt{throw}\;\; x;\}\;\; \omega\rangle\phi, \Delta}{\Gamma \vdash \langle \pi \;\; \texttt{try}\{\texttt{throw}\;\; x;\;\; a\}\;\; \texttt{finally}\{f\}\;\; \omega\rangle\phi, \Delta}$$

1em] Note that the execution of the `try` block, which might have been a former `catch` clause, is stopped immidately and the `finally` block is executed before the exception is thrown.

Chapter 4

Comparison of the two Calculi

As we compare an implementation of a calculus with a version on paper of a different calculus there are a lot of differences between them. The Bali calculus for example uses Isabelle functions in difference to the KeY calculus, which is due to the version on paper free of any system dependent functions or datatypes. One of the advantages of a realization of a calculus in a proving system is that the calculus can be tested automatically and properties of the calculus may directly be proved. This advantage of Bali, is the reason why we try to map the KeY calculus to the Bali calculus.

In the following sections, aspects of the calculi are discussed, such as the context, the actual JAVA programs or the way exceptions are handled in both calculi.

4.1 Context

What is named a context in both calculi is quite similar. A KeY context covers the class and interface definitions, which are used in the current program, as well as the program to be verified itself. The Bali context only contains the class and interface definitions, which belong to the program, that is the subject of the verification.

4.2 Java Syntax

The JAVA part of formulas in the calculi is similar in general in the calculi. The KeY calculus covers the whole range of JAVA CARD statements and displays them in the original JAVA syntax. A few restrictions are made on the number of statements covered in Bali. For example only one of the loop statements is part of the calculus. The Bali calculus attaches annotations to some statements, which provide information of the kind a compiler would produce. Additionally the JAVA syntax in Bali is slightly changed due to the embedding in Isabelle/HOL. Because of some generalisations that have been made during the mapping of JAVA statements to Isabelle dataypes the set of valid JAVA statements has been increased.

4.3 Exception Handling

Exceptions are handled differently in both calculi. The KeY calculus has a rule for every combination of statement and reason for abrupt completion. If the statement of a rule is of the appropriate type to catch an exception, the exception is caught and disappears. Otherwise the statement is skipped and the reason for abrupt completion meets the next enclosing statement. If an exception cannot be caught by any of the enclosing statements, the last version of the program contains no other statements than the exception and it is said that the program terminates abruptly.

In the Bali calculus, exceptions are either caught in the rule in which they were raised, or they are added to the final state of a judgment. If this happens, there is a rule which states that if the initial state contains an exception, the final state is equal to the initial state, that is the current statement is skipped.

4.4 Additional functions

In order to keep track of the current value of objects, fields and variables without an explicit state with access functions, the KeY calculus defines update functions (see section 3.7). An update function is attached to all parts of the sequent that is affected by a value change of a variable.

Chapter 5

Generation of Proof Obligations

Despite of all differences between the two calculi we try to make use of the Bali calculus to proof the correctness of our rules. As the KeY calculus contains about one hundred rules, it is highly desirable to use a theorem prover for this task. This requires to embed the types and a few properties of JAVA into the prover. To avoid this work we chose to adjust the syntax of our rules to the syntax of the Bali calculus so that we can use the already existing JAVA environment of Bali in Isabelle.

5.1 General idea

In order to develop an appropriate adjustment, we need to look at the input format of the KeY rules and the required output format.

The general scheme of a KeY sequent looks like this: 0.7em] $\Gamma, \mathcal{U}\gamma \vdash \mathcal{U}\langle\pi p\omega\rangle\phi, \Delta$ 0.7em] where

- Γ is a conjuction of DL formulas which may contain programs, together with $\mathcal{U}\gamma$ Γ contains the preconditions for the execution of the program $\pi p\omega$
- $\mathcal{U}$ is an update function of the form $\{x \leftarrow t\}$ which assigns term t to variable x
- γ is a formula in first-order logic, that is in all cases an equation without programs.
- $\pi p\omega$ is the current program, where p is the first executable statement
- ϕ is the postcondition of the program which has to be satisfied by the final state
- Δ is a disjunction of formulas of the structure $\mathcal{U}\langle\pi p\omega\rangle\phi$

A general Bali judgment $G \vdash s_0 - p_B \to s_f$ consists of

- G which contains of the definition of all classes and interfaces of the context
- s_0 which is the initial state for a run of the program
- p_B is the program that is to be executed
- s_f is the final state, reached after executing p_B

The result of this mapping is supposed to be used as a proof obligation for proofs in Isabelle. For such a proof the premises and the conclusion lo a KeY rule is mapped into Bali syntax. Then Isabelle is used to prove that all initial states which satisfy the premises satisfy the conclusion, too. To make such a proof possible, the mapping requires to have the following properties:

1. The set of initial states which satisfies the premises of a rule in KeY syntax has to satisfy the premises in the Bali syntax.
2. 1. has to be true for the conclusion of a rule
3. A KeY program and its equivalent in Bali syntax may not reach different final states when started in the same initial state.

Next we examine each part of a general sequent in KeY syntax to find out if it needs to be mapped or can be omitted and the properties listed above are still preserved.

Γ, the first symbol in a KeY sequent, reduces the number of possible initial states as all states, which possibly satisfy the sequent have to satisfy Γ. Γ is only a scheme which is not further specified in any of the rules, and it is not changed in any of the rules. Therefore it is not necessary to include Γ in the mapping. Without Γ the sequent might be satisfied by more states, but the original subset of states will be still included.

Only sequents in which the $\langle\pi\ \mathtt{p}\ \omega\rangle\phi$ part can be satisfied are considered as input. Therefore Δ does not need to be included, because if the formula $\langle\pi\ \mathtt{p}\ \omega\rangle\phi$ is already valid, there is no need to evaluate an addition formula which is connect to it with a disjunction.

Updates are not necessary for the mapping as long as the same updates are attached to the program and γ. Updates keep track of the changed value of variables due to an assignment, but as we consider only arbitrary states which satisfy γ there is no need to know exact values. Updates only need to be mapped if the premises of a rule contains different updates than the conclusion. Without the updates the information about the difference in the satisfying states is lost and the rule is not correct anymore. Fortunately different updates in premises and conclusion occur only in the assignment rule. For sake of simplicity we do not consider the assignment rule here and may therefore omit to map updates.

Another simplification is to use a reduced form of the KeY programs. This is possible because no rule exists where π and ω are changed by the rule. Hence they cannot affect the properties listed above and may be omitted.

After these reductions only γ and p need to be mapped as they are the parts of the rule which change. They have to be rewritten as s_0, s_f and p_B.

So what we need is a function τ of the kind: $\langle\gamma, p\rangle \rightarrow \langle S(x), p_B\rangle$ where

- γ is a first-order formula without programs
- $\mathcal{U}$ is an update function of the form $\{t \leftarrow e\}$
- $S(x)$ is a predicate which expresses that state x satisfies the condition γ
- p is a Java Card program

For all instances $\langle\overline{\gamma}, \overline{p}\rangle$ and $\langle\overline{s}, \overline{p}_B\rangle$ of $\langle\gamma, p\rangle \rightarrow \langle S(x), p_B\rangle$ the following is supposed to be true: All initial states s_0 and final states s_f have the following properties:

- $s_0 \models \gamma$
- the execution of p terminates in s_f
- $s_f \models \phi$

if and only if s_0 and s_f satisfy the following condition:
IF S(s_0) THEN $\vdash s_0 - p_B \rightarrow s_f$ is derivable in Bali.

Up to now we have been talking about a mapping τ without exactly specifying it. A close look at the signature $\langle\gamma, p\rangle \rightarrow \langle S(x), p_B\rangle$ of τ reveals that τ consists of two mappings: the first one maps γ to $S(x)$, the second one changes the syntax of p into p_B, that is from standard Java syntax to the Bali syntax. A description of this mapping is implicitly included in the explanations of the Bali datatypes and can be found in chapter 2 and [12].

The mapping of γ to $S(x)$ is more complicated. An inspection of the KeY rules reveals two possible structures of γ: either γ is empty or it is a logical equality between a variable and a value. In case γ is empty, $S(x)$ returns true as every state satisfies an empty γ. In case γ is an equation, $S(x)$ tests if the variable in the equation contains in state x the specified value. This is done via an Isabelle if statement, which has a condition that fetches the value of the variable from the table of local variables of state x and tests if that value is equal to the one specified in γ.

Now we have all necessary ingredients to generate the proof obligations from the KeY rules. The next section shows a list of the KeY rules and their equivalent in Bali syntax. An example for a proof that the KeY rules are correct with respect to the Bali formalisation is shown in chapter 6.

5.1.1 Rules in Bali syntax

Due to the limitations of the Bali calculus, some KeY rules are shown in a simplified version. All rules in Bali syntax are adjusted to the syntax of the new Bali version.

Some rules contain functions which are not part of Bali. They can be found in the appendix.

transformation-?:-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \texttt{if(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ \ cnd \ \texttt{?} \ a \ \texttt{:} \ b \ \omega \rangle \phi, \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ (\mathsf{Expr}\ a)\ \mathsf{Else}\ (\mathsf{Expr}\ b) \ \rightarrow\ s_1}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}(cnd\ ?\ a\ \ :\ \ b)\ \rightarrow\ s_1}$$

This rule performs a mere program transformation. The Expr prefix for the expressions a and b is needed to build a statement out of the expression a

if-condition-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \ bv \ \texttt{=} \ cnd\texttt{;} \ \texttt{if(}bv\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \ \texttt{if} \ \texttt{(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}$$

Bali syntax:

$$\frac{\begin{array}{c}\mathsf{not\_in\_expr}\ bv\ cnd\ \wedge\ \mathsf{not\_in\_stmt}\ bv\ a\ \wedge\ \mathsf{not\_in\_stmt}\ bv\ b\ \wedge \\ G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (\mathsf{LVar}\ (\mathsf{EName}\ bv)\ \ :=\ \ cnd);;\ \mathsf{If}\ (\mathsf{Acc}\ (\mathsf{LVar}\ (\mathsf{EName}\ bv)))\ a\ \mathsf{Else}\ b\ \rightarrow\ s_1\end{array}}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_1}$$

not_in_expr and not_in_stmt are functions which return true if variable bv is not contained in the specified expression or statement. Additionally the translated version of the rule assumes, that the variable bv of type boolean is already included in the state.

if-right0.7em] *KeY:*

$$\frac{\Gamma, cnd \doteq true \ \vdash \ \langle \pi \ \texttt{a} \ \omega \rangle \phi, \ \Delta \qquad \Gamma, cnd \doteq false \ \vdash \ \langle \pi \ \texttt{b} \ \omega \rangle \phi, \ \Delta}{\Gamma \ \vdash \ \langle \texttt{if(}cnd\texttt{)} \ \texttt{a else b} \ \omega \rangle \phi, \ \Delta}$$

Bali syntax:

$$\frac{\begin{array}{c}(G \vdash \mathsf{Norm}\ s_0 - a\ \rightarrow\ s_1) \wedge (\mathsf{the\_Bool}(\mathsf{ev\_boolvar}\ cnd\ s_0\ \ G)) \\ (G \vdash \mathsf{Norm}\ s_0 - b\ \rightarrow\ s_1) \wedge \neg(\mathsf{the\_Bool}(\mathsf{ev\_boolvar}\ cnd\ s_0\ \ G))\end{array}}{(G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_1)}$$

This is the general `if` rule. The function ev_boolvar evaluates the value of an expression in a certain state s in a context G. The boolean value of the returned type val is calculated by the_Bool.

if-positive-right0.7em] *KeY:*

$$\frac{\Gamma \ \vdash \ cnd \doteq true \qquad \Gamma \ \vdash \ \langle \pi \ \ a \ \ \omega \rangle \phi, \Delta}{\Gamma \ \vdash \ \langle \pi \ \texttt{if} \ \texttt{(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega \rangle \phi, \Delta}$$

Bali syntax:

$$\frac{\mathsf{the\_Bool}(\mathsf{ev\_boolvar}\ cnd\ s_0\ \ G) \qquad G \vdash \mathsf{Norm}\ s_0 - a\ \rightarrow\ s_1}{\exists b.\ G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_1}$$

Once we can determine that the `if` condition is true, the execution of the `else` branch can be omitted.

if-negative-right0.7em] Analogous to the if-positive-right rule 1.0em]

The syntax of the Bali calculus provides try statements with only one catch clause in contrast to an arbitrary number of `catch` clauses in the KeY calculus. This gives two possibilities to express KeY rules in Bali syntax: a) the additional `catch` clauses are simulated by nested If statements or the InstOf operator, b) the KeY rules are restricted to one `catch` clause. Here the second option is chosen to maintain a better readability.

tc-throw-catch-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \texttt{if(}x\ \texttt{instanceof}\ E\texttt{)}\ \{e\ \texttt{=}\ x\texttt{;}\ b\}\ \texttt{else}\ \{\texttt{throw}\ x\texttt{;}\}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi \ \texttt{try}\{\texttt{throw}\ x\texttt{;}\ a\}\ \texttt{catch(}E\ e\texttt{)}\{b\}\ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(x\ \mathsf{InstOf}\ (\mathsf{ClassT}\ E))\ (\mathsf{Expr}\ (\mathsf{LVar}\ (\mathsf{EName}\ (\mathsf{VNam}\ e)) := x);;\ b)\ \mathsf{Else}\ \mathsf{Throw}\ x \rightarrow s_1}{\exists a.\ G \vdash \mathsf{Norm}\ s_0 - \mathsf{Try}\ (\mathsf{Throw}\ x;;\ a)\ \mathsf{Catch}(E\ e)\ b \rightarrow s_1}$$

This rule is used to show that a certain exception is caught by the `catch` clause. Expr transforms the assignment expression into a statement, which is needed for the concatenation operator ;;. LVar EName VNam casts the type of e step by step into the appropriate supertype of a local variable, the type of the left side of an assignment.

tc-try-empty-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi \ \texttt{try}\{\}\ \texttt{catch(}E\ e\texttt{)}\{b\}\ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Skip} \rightarrow s_1}{\exists E\ e\ b.\ G \vdash \mathsf{Norm}\ s_0 - \mathsf{Try}\ \mathsf{Skip}\ \mathsf{Catch}(E\ e)b \rightarrow s_1}$$

If no reason for abrupt completion occurs during the execution of the Try block, the block is empty in the end and has to be deleted. As the Try block is syntactically not allowed to be empty in the Bali calculus, the empty statement Skip is used instead. 1.0em]

A `try-catch` statement with a `finally` block is in Bali modelled as a combination of a Try-Catch statement and a Finally statement. The Finally statement takes two arguments, where the first one is the Try-Catch statement and the second argument contains the actual Finally block.

tcf-throw-catch-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \texttt{try}\{\ \texttt{if(}x\ \texttt{instanceof}\ E\texttt{)}\ \{e\ \texttt{=}\ x\texttt{;}\ b\}\ \texttt{else}\ \{\texttt{throw}\ x\texttt{;}\}\}\ \texttt{finally}\ c\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi \ \texttt{try}\{\texttt{throw}\ x\texttt{;}\ a\}\ \texttt{catch(}E\ e\texttt{)}\{b\}\ \texttt{finally}\ c\ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - (\mathsf{If}(x\ \mathsf{InstOf}\ (\mathsf{ClassT}\ E))\ (\mathsf{Expr}\ (\mathsf{LVar}\ (\mathsf{EName}\ (\mathsf{VNam}\ e)) := x);;\ b)\ \mathsf{Else}\ \mathsf{Throw}\ x)\ \mathsf{Finally}\ c \rightarrow s_1}{\exists a.\ G \vdash \mathsf{Norm}\ s_0 - (\mathsf{Try}\ (\mathsf{Throw}\ x;;\ a)\ \mathsf{Catch}(E\ e)b)\ \mathsf{Finally}\ c \rightarrow s_1}$$

Analogous to the tc-throw-catch right rule.

tcf-throw-nocatch-right0.7em] *KeY:*

$$\frac{\Gamma \vdash \langle \pi \ \{c\texttt{;}\ \texttt{throw}\ x\}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \langle \pi \ \texttt{try}\{\texttt{throw}\ x\texttt{;}\ a\}\ \texttt{finally}\ c\ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - c;;\ \mathsf{Throw}\ x\ \rightarrow\ s_1}{\exists a.\ G \vdash \mathsf{Norm}\ s_0 - (\mathsf{Throw}\ x;;\ a)\ \mathsf{Finally}\ c\ \rightarrow\ s_1}$$

The language specification allows `try-catch-finally` statements without `catch` block. As Bali does not allow to use the Try statement without a Catch clause, this statement is modelled as a Finally statement which contains the `try` block as a composition of statements as first argument. Of course, then the key word Try does not appear.

tcf-try-empty-right0.7em] *KeY:*

$$\frac{\Gamma\ \vdash\ \langle\pi\ \ c\ \ \omega\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\pi\ \texttt{try}\{\ \}\ \texttt{catch}(E\ e)\{b\}\ \texttt{finally}\ \ c\ \ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - c\ \rightarrow\ s_1}{\exists E\ \ e\ \ b.\ G \vdash \mathsf{Norm}\ s_0 - (\mathsf{Try\ Skip\ Catch}(E\ \ e)\ \ b)\ \mathsf{Finally}\ \ c\ \rightarrow\ s_1}$$

Analogous to the tc-try-empty-right rule. 1.0em]

As Bali does not support labels, `continue` or `break`, only a reduced version of the KeY `while` rule is shown here.

while-right0.7em] *KeY:*

$$\frac{\Gamma\ \vdash\ \langle\pi\ \ \texttt{if}(cnd)\ \ \{a\ \texttt{while}\ \ (cnd)\ \ \texttt{a}\}\ \ \omega\rangle\phi, \Delta}{\Gamma\ \vdash\ \langle\pi\ \ \texttt{while}(cnd)\ \ a\ \ \omega\rangle\phi, \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ (a;;\ \ l:\ \ \mathsf{While}\ (cnd)\ \ a)\ \mathsf{Else\ Skip}\ \rightarrow\ s_1}{G \vdash \mathsf{Norm}\ s_0 - l:\ \ \mathsf{While}\ (cnd)\ \ a\ \rightarrow\ s_1}$$

This rule unwinds a While loop one time and places it in an If statement to ensure that it is only executed if the condition is true. As Bali does not support If statement without Else block the Else block contains the empty statement Skip. 1.0em]

A `throw` statement has an expression as argument, which specifies the exception to throw. The KeY calculus requires that this expression is evaluated separately before the exception is thrown. Here is the rule which evaluates the expression:

throw-evaluation-right0.7em] *KeY:*

$$\frac{\Gamma\ \vdash\ \langle\pi\ \ x\ \texttt{=}\ a\texttt{;}\ \ \texttt{throw}\ \ x\ \ \omega\rangle\phi,\ \Delta}{\Gamma\ \vdash\ \langle\pi\ \ \texttt{throw}\ \ a\texttt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Bali syntax:

$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (\mathsf{LVar}\ (\mathsf{EName}\ ev))\ \ :=\ \ a;;\ \ \mathsf{Throw}\ (\mathsf{Acc}\ (\mathsf{LVar}\ (\mathsf{EName}\ ev)))\ \rightarrow\ s_1}{G \vdash \mathsf{Norm}\ s_0 - \mathsf{Throw}\ \ a\ \rightarrow\ s_1}$$

LVar EName casts the name of the local variable *ev* into the general type of a local variable which is required for the left side of an assignment. Expr transforms the assignment expression into a statement. Acc is the necessary operator to cast the local variable with name *ev* into an access to a variable expression, the Throw statement requires an expression as argument.

Chapter 6

Correctness Proof of a Rule

Since both calculi use sequents, we first define the notion of a derivable sequent and then the notion of a derivable rule.

A sequent $\Gamma \vdash \Delta$ is derivable iff: (a) $\Gamma \vdash \Delta$ is an instance of a rule (axiom) (b) there is an instance of a rule ρ such that $\Gamma \vdash \Delta$ is the conclusion of this instance.

A rule ρ with premisses $\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n$ and conclusion $\Gamma \vdash \Delta$ is derivable, if all premisses are derivable and there is a derivation of sequents from the premisses to the conclusion.

Let us assume the generation of the Bali syntax of KeY rules is correct. If we can show that a KeY rule in Bali syntax is derivable in the Bali calculus we have shown that this rule is correct, as it is only an abbreviation of a valid derivation in this calculus. This conclusion is correct, as the operational sematics of the Bali formalization, which we use for the derivation, has been shown correct with respect to the axiomatic semantics of the Bali formalization.

Hence to prove a rule ρ_B in Bali syntax correct with respect to the Bali formalization, we need to show that the conclusion of ρ_B is derivable from the premisses of ρ_B. To show this derivability we may use the operational semantics of the Bali formalization

The following sections show the proof for the `?:-right` rule as an example.

6.1 Proof of `?:`-right

The `?:` operator is the expression equivalent to an if statement. As an expression is one of all possible statements, the `?:` expression in particular can be transformed into a statement, which is an `if` statement in this case. This transformation is the task of the `?:`-right rule shown in Fig. 6.1.

In order to prove this rule correct, two special rules are needed. They handle an If statement and an expression statement in premises. These rules are neccessary because we intend to show that the conclusion of the *if-transformation-right* rule is derivable from the premises, so we need to be able to handle all parts which occur in the premises. The two rules are developed in the

$$\text{"}(?:-right)\text{"} \quad \frac{\Gamma \vdash \langle \pi \ \texttt{if} \ \texttt{(}cnd\texttt{)} \ a \ \texttt{else} \ b \ \omega\rangle\phi, \Delta}{\Gamma \vdash \langle \pi \ \texttt{(}cnd\texttt{)?}a\texttt{:}b \ \omega\rangle\phi, \Delta}$$

$$\text{"}(if-transformation-right_{Bali})\text{"} \quad \frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)(\mathsf{Expr}\,a)\mathsf{Else}(\mathsf{Expr}\,b) \rightarrow s_1}{G \vdash \mathsf{Norm}\ s_0 - \texttt{Expr}\ (cnd?a\texttt{:}b) \rightarrow s_1)}$$

Figure 6.1: Original `?:`-right rule and its equivalent in Bali syntax

following two subsections, followed by another subsection which contains the main proof.

6.1.1 Reverse if rule

The Bali calculus provides a rule for If statements which contains the If statement in the conclusion. This rule is part of the inductive definition of the set eval G, which contains quadruples of the kind (*initial state, term, return value, final state*). If the existing rule could be reversed, we would have the required rule. This is expressed in the following lemma:

Lemma 6.1.1
$$\frac{G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_2}{\begin{array}{ll}\exists v\ s1.\ (& G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow s_1 \wedge \\ & G \vdash \mathsf{s}_1 - \mathsf{if}(\mathsf{the\_Bool} v)\ \mathsf{then}\ a\ \mathsf{else}\ b\ \rightarrow\ s_2)\end{array}}$$

To proof this lemma we can make use of an instantiation of the so called 'cases' rule, which Isabelle generates for every inductive definition of a set. Such a cases rule has the premises that a certain variable is element of the set and that, for every possible structure of this element a certain predicate is true. The conclusion of the cases rule contains only the predicate.

An instantiation of this rule means to chose a certain structure for the variable. This reduces the premises of the cases rule to two: The first one states that the chosen structure is an element of the set and the second states that this structure implies a certain predicate. Using this new rule as an elimination rule returns only one subgoal instead of a number of subgoals equal to the number of rules that construct the inductive set.

For the reversed If rule, the instance of the cases rule is generated via

inductive_cases if_reverse: "(Norm s_0, In1r (If(cnd) a Else b), dummy_res, s_1) $\in$ eval G"

which produces a new rule called `if_reverse`, which has the specified quartuple as premiss:

$$\begin{array}{l}[\!|\ G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_2; \\ \quad \bigwedge c\ d\ ca.\ [\!|G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow c \rightarrow (d, ca); \mathsf{the\_Bool}\ c;\ G \vdash (d, ca) - a\ \rightarrow\ s_2\ |\!] \\ \qquad\qquad \Longrightarrow P; \\ \quad \bigwedge c\ d\ ca.\ [\!|G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow c \rightarrow (d, ca); \sim \mathsf{the\_Bool}\ c;\ G \vdash (d, ca) - a\ \rightarrow\ s_2|\!] \\ \qquad\qquad \Longrightarrow P|\!] \\ \Longrightarrow P\end{array}$$

Now we are prepared to prove the reversed If rule as lemma `rev_if` (see Fig. 6.2):

$$\begin{array}{l}\text{lemma rev\_if: "}[\!|G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ a\ \mathsf{Else}\ b\ \rightarrow\ s_2|\!] \Longrightarrow \\ \qquad\qquad \exists\ v\ s1.\ ((G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow s_1) \wedge \\ \qquad\qquad\qquad (G \vdash s_1 - \mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ a\ \mathsf{else}\ b\ \rightarrow\ s_2))\text{"}\end{array}$$

The first step in the proof is to apply the `if_reverse` rule, which results in two subgoals (Fig. 6.3). Simplifying all subgoals using `simp_all` stops at an existential quantifier in both subgoals (Fig. 6.4). Both quantifiers need be removed using the exI rule (introduction of an existential quantifier) (Fig. 6.5 and 6.6). The rest of the proof can then be done by blast which is a tableau prover.

6.1.2 Reverse expression rule

As the premiss of the main lemma contains an expression statement, a reverse expression rule (with an expression statement in the premisses) is needed for the proof. This rule is obtained in the same way as the reverse if rule.

The first step is to generate an instance of the cases rule which belongs to the inductive set eval, with the variable instanciated with a quartuple that contains an expression statement

```
inductive_cases if_reverse: "(Norm s0, In1r (If(cnd) a Else b), dummy_res, s2). eval G"

lemma rev_if: "[|G ⊢ Norm s0 − If(cnd) a Else b → s2|] ⟹
∃ v s1. ((G ⊢ Norm s0 − cnd −≻ v → s1) ∧
          (G ⊢ s1 − if the_Bool v then a else b → s2))"
apply(erule if_reverse)
apply(simp_all)
    apply(rule ex_I, blast)+
done
```

Figure 6.2: Proof of the reversed If rule

```
inductive_cases if_reverse: "(Norm s0, In1r (If(cnd) a Else b), dummy_res, s2). eval G"

lemma rev_if: "[|G ⊢ Norm s0 − If(cnd) a Else b → s2|] ⟹
∃ v s1. ((G ⊢ Norm s0 − cnd −≻ v → s1) ∧
          (G ⊢ s1 − if the_Bool v then a else b → s2))"

apply(erule if_reverse)

1. ⋀ba aa baa aaa bb.
         [|s2 = (aaa, bb);  G ⊢ Norm s0 − cnd −≻ ba → (aa, baa); the_Bool ba;
           G ⊢ (aa, baa) − a → (aaa, bb)|]
         ⟹ ∃v s1. G ⊢ Norm s0 − cnd −≻ v → s1 ∧
                G ⊢ s1 − if the_Bool v then a else b → s2
2. ⋀ba c1 c2 e s0a aa baa aaa bb.
         [|s0 = s0a;  s2 = (aaa, bb);  G ⊢ Norm s0 − e −≻ ba → (aa, baa); ∼the_Bool ba;
           cnd = e;  a = c1;  b = c2;  G ⊢ (aa, baa) − c2 → (aaa, bb)|]
         ⟹ ∃v s1. G ⊢ Norm s0 − cnd −≻ v → s1 ∧
                G ⊢ s1 − if the_Bool v then a else b → s2
```

Figure 6.3: First step of the proof of the reverse If rule

⋮

apply(simp_all)

1. $\bigwedge ba\ aa\ baa\ aaa\ bb.$
   $[|s_2 = (aaa, bb);\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail ba \rightarrow (aa, baa);\ \mathsf{the\_Bool}\ ba;$
   $G \vdash (aa, baa) - a \rightarrow (aaa, bb)|]$
   $\Longrightarrow \exists v.\ (\mathsf{the\_Bool}\ v \longrightarrow$
   $(\exists aa\ b.\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail v \rightarrow (aa, b) \wedge$
   $G \vdash (aa, b) - a \rightarrow (aaa, bb))) \wedge$
   $(\sim \mathsf{the\_Bool}\ v \longrightarrow$
   $(\exists a\ ba.\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail v \rightarrow (a, ba) \wedge$
   $G \vdash (a, ba) - b \rightarrow (aaa, bb)))$
2. $\bigwedge ba\ c1\ c2\ e\ s_0a\ aa\ baa\ aaa\ bb.$
   $[|s_0 = s_0a;\ s_2 = (aaa, bb);\ G \vdash \mathsf{Norm}\ \mathsf{s_0a} - e \rightarrowtail ba \rightarrow (aa, baa);\ \sim \mathsf{the\_Bool}\ ba;$
   $cnd = e;\ a = c1;\ b = c2;\ G \vdash (\mathsf{aa}, \mathsf{baa}) - c2 \rightarrow (aaa, bb)|]$
   $\Longrightarrow \exists v.\ (\mathrm{the\_Bool}\ v \longrightarrow$
   $(\exists a\ b.\ G \vdash \mathsf{Norm}\ \mathsf{s_0a} - e \rightarrowtail v \rightarrow (a, b) \wedge$
   $G \vdash (\mathsf{a}, \mathsf{b}) - c1 \rightarrow (aaa, bb))) \wedge$
   $(\sim \mathsf{the\_Bool}\ v \longrightarrow$
   $(\exists a\ b.\ G \vdash \mathsf{Norm}\ \mathsf{s_0a} - e \rightarrowtail v \rightarrow (a, b) \wedge$
   $G \vdash (\mathsf{a}, \mathsf{b}) - c2 \rightarrow (aaa, bb)))$

Figure 6.4: Simplified subgoals after applying simp_all

⋮

apply(rule exI) (* is only applied to first subgoal *)

1. $\bigwedge ba\ aa\ baa\ aaa\ bb.$
   $[|s_2 = (aaa, bb);\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail ba \rightarrow (aa, baa);\ \mathsf{the\_Bool}\ ba;$
   $G \vdash (aa, baa) - a \rightarrow (aaa, bb)|]$
   $\Longrightarrow (\mathsf{the\_Bool}\ (?v16\ ba\ aa\ baa\ aaa\ bb) \longrightarrow$
   $(\exists ab\ b.\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail ?v16\ ba\ aa\ baa\ aaa\ bb \rightarrow (ab, b) \wedge$
   $G \vdash (ab, b) - a \rightarrow (aaa, bb))) \wedge$
   $(\sim \mathsf{the\_Bool}\ ?v16\ ba\ aa\ baa\ aaa\ bb \longrightarrow$
   $(\exists a\ bc.\ G \vdash \mathsf{Norm}\ s_0 - cnd \rightarrowtail ?v16\ ba\ aa\ baa\ aaa\ bb \rightarrow (a, bc) \wedge$
   $G \vdash (a, bc) - b \rightarrow (aaa, bb)))$

apply(blast)

Figure 6.5: End of first subgoal of proof of the reverse If rule

⋮
apply(rule exI)

2. $\bigwedge ba\ c1\ c2\ e\ s_0a\ aa\ baa\ aaa\ bb.$
$[|s_0 = s_0a;\ s_2 = (aaa, bb);\ G \vdash \mathsf{Norm}\ s_0a - e \rightarrowtail ba \rightarrow (aa, baa);\ \mathsf{the\_Bool}\ ba;$
$cnd = e;\ a = c1;\ b = c2;\ G \vdash (aa, baa) - c2 \rightarrow (aaa, bb)|]$
$\Longrightarrow (\mathsf{the\_Bool}\ (?v49\ ba\ v1\ c2\ e\ s_0a\ aa\ baa\ aaa\ bb) \longrightarrow$
$(\exists a\ b.\ G \vdash \mathsf{Norm}\ s_0a - e \rightarrowtail ?v49\ ba\ c1\ c2\ s_0a\ aa\ baa\ aaa\ bb \rightarrow (a, b) \wedge$
$G \vdash (a, b) - c1 \rightarrow (aaa, bb))) \wedge$
$(\sim \mathsf{the\_Bool}\ ?v49\ ba\ c1\ c2\ e\ s_0a\ aa\ baa\ aaa\ bb \longrightarrow$
$(\exists a\ b.\ G \vdash \mathsf{Norm}\ s_0a - e \rightarrowtail ?v49\ ba\ c1\ c2\ e\ s_0a\ aa\ baa\ aaa\ bb \rightarrow (a, b) \wedge$
$G \vdash (a, b) - c2 \rightarrow (aaa, bb)))$

apply(blast)

Figure 6.6: End of second subgoal of proof of the reverse If rule

(Fig. 6.7). This instance is applied as first step to the lemma rev_expr, which contains the actual reversed rule. In contrast to the reverse If proof the application of expr_reverse does not split the goal. Next we remove the equations from the premisses using safe. Then the premiss can not further be simplyfied, so we apply exI to remove the existential quantifier from the conclusion. Then we end up in a state, where premiss and conclusion are equal apart from a substitution, which has not been done yet. Isabelle finds the right substitution itself if assumption is invoked and the reversed expression rule is proven correct.

inductive_cases Expr_reverse: "(Norm $s0$, In1r (Expr e), dummy_res, $s2$) $\in$ eval G"

lemma rev_expr: "$G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ e \rightarrow s_1 \Longrightarrow \exists\ v.\ G \vdash \mathsf{Norm}\ s_0 - e \rightarrowtail v \rightarrow s_1$"
apply(erule expr_reverse)
apply(simp_all)
  apply(rule exI)
  apply(blast)
done

Figure 6.7: Proof of the reverse Expr rule

6.1.3 Main proof

Having finished the preparations, we can start to prove the main lemma (see Fig. 6.8):

Lemma 6.1.2 lemma iftransformation: "$G \vdash \mathsf{Norm}\ s_0 - \mathsf{If}(cnd)\ (\mathsf{Expr}\ a)\ \mathsf{Else}\ (\mathsf{Expr}\ b) \rightarrow s_1 \Longrightarrow G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (cnd?a : b) \rightarrow s_1$"

The best tactic is to work first on the premiss, as then first the schematic variables for premiss are generated. If later the conclusion needs to be unified so that a formula of the premiss is equivalent to a formula from the conclusion, the variables of the conclusion can only be substituted with already existing variables from the premiss. If we would first work on the conclusion and then on the premiss, we would end up not being able to unify both.

Inspection of the premiss reveals that there is only one way to start the proof which makes sense: to apply the rev_if rule in a destructive way. This results in replacing the former premiss with the conclusion of the applied rule:

$$\begin{aligned}\exists\, v\ s_1 a.\ \ & G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow s_1\, a \\ & \wedge\ G \vdash \mathsf{s_1a} - \mathsf{if\ the\_Bool}\ v\ \mathsf{then\ Expr}\ a\ \mathsf{else\ Expr}\ b\ \rightarrow\ s_1 \\ & \Longrightarrow G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (cnd?a : b)\ \rightarrow\ s_1\end{aligned}$$

Before any other rule can be applied to the premisses, the existential quantifier needs to be removed. This and a few simplifications can be done using the method safe. Next we need to work on the second premiss. As the first premiss may throw an exception, the second one may or may not start in an exception free state. As there is no rule to handle such an ambigous state, we need to split the proof in two cases. Having done this safe and simp simplify the first subgoal. simp splits the Isabelle if of the second formula in the premiss and therefore generates two subgoals out of the first one.

1. $\bigwedge v\ ba.\ [|G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow \{sfNorm\ ba;\ \mathsf{the\_Bool}\ v;\ G \vdash \mathsf{Norm\ ba} - \mathsf{Expr}\ a\ \rightarrow\ s_1|]$
$\Longrightarrow G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}(cnd?a : b)\ \rightarrow\ s_1$

2. $\bigwedge v\ ba.\ [|G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow \mathsf{Norm}\ ba;\ \sim \mathsf{the\_Bool}\ v;\ G \vdash \mathsf{Norm}\ ba - \mathsf{Expr}\ a\ \rightarrow\ s_1|]$
$\Longrightarrow G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (cnd?a : b)\ \rightarrow\ s_1$

As both subgoals differ only in the value of the_Bool v, they can be solved in the same way. First $G \vdash \mathsf{Norm}\ ba - \mathsf{Expr}\ a\ \rightarrow\ s_1$ needs to be simplified with the reverse expression rule rev_expr. Then safe removes the exists quantifier which appears after the expression rule application. Now the premisses are as simple as possible. The conclusion $G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (cnd?a : b)\ \rightarrow\ s_1$ allows only one sensible rule application, the Expr rule, which deletes Expr and includes a return value for a or b. Next the `?:` operator is removed by applying the Cond rule. This rule splits the subgoal into two parts, one for the case where the condition cnd ist true and one for the case where cnd is false. The first case can be closed by assumption, i.e. premisses and conclusion contain at least one equal formula, and the second one can be closed using the simplifier simp.

Having reached this state, only one subgoal is left:

$$\begin{aligned}\bigwedge v\ aa\ ba\ aaa.\ \ & [|G \vdash \mathsf{Norm}\ s_0 - cnd \succ\!\!\rightarrow v \rightarrow (\mathsf{Some}\ aaa,\ ba); \\ & G \vdash (\mathsf{Some}\ aaa,\ ba) - \mathsf{if\ the\_Bool}\ v\ \mathsf{then\ Expr}\ a\ \mathsf{else\ Expr}\ b\ \rightarrow\ s_1|] \\ & \Longrightarrow G \vdash \mathsf{Norm}\ s_0 - \mathsf{Expr}\ (cnd?a : b)\ \rightarrow\ s_1\end{aligned}$$

Here no rule can be applied to the second formula of the premisses as is starts in a state with an exception. The conclusion is the same as above, so the Expr and Cond rules are applied. Again, the later splits the subgoal into two cases. Both can be solved by the tableaux prover blast, which finishes the proof.

```
lemma iftransformation: "G ⊢ Norm s0 − If(cnd) (Expr a) Else (Expr b) → s1
                        ⟹ G ⊢ Norm s0 − Expr (cnd?a : b) → s1"
apply(drule rev_if)
apply(safe)
apply(case_tac aa)
    apply(safe, simp)
        apply(drule rev_expr, safe, rule Expr, rule Cond, assumption, simp)+
apply(rule Expr)
apply(rule Cond)
    apply(assumption)
    apply(simp)
        apply(blast)+
done
```

Figure 6.8: Proof of the *if-transformation-right*$_{Bali}$ rule

Chapter 7

Conclusion

One rule of the KeY calculus has been shown correct with respect to the Bali formalisation of Java for Isabelle.

The mapping of KeY rules into the Bali syntax required a good understanding of the basic concepts of both calculi such as state and context, to omit the mapping of unnecessary information.

Learning to handle Isabelle proved to be more time demanding as expected because Isabelle is a complex system with more than 700 pages of documentation. Without having an expert to ask at hand the introduction would be even more time consuming, as very helpful mailing list exists, but asking more than one or two questions a day is in general impossible. Furthermore the usual problems of a beginner are of a kind which can be solved very easily, a glance at the error message of a trained Isabelle user is normally sufficient to solve the problem.

Once this stage is mastered, the system can be used to type check Java statements in Bali which usually discovers a lot of errors, as the Bali syntax contains a lot of nested constructs so that one often forgets a surrounding constructor. In order to prove the retrieved translations correct, the interaction with the system demands a good knowledge of proof tactics in Isabelle. For an untrained person assuming a mere knowledge of the necessary commands results in a trial and error process with no guarantee to succeed.

Once one proof is done, further proofs can be constructed following the same scheme and are therefore a lot easier and faster to handle. Writing the initial proof, a simple program transformation with only two constructors, took about one and a half weeks. Further proofs following the same principle should not take longer than one or two days.

7.1 Further Work

One rule of a calculus proved correct does not make the whole calculus correct. The correctness proofs for the remaining rules shown in this paper should work the same way as the example. This requires the existence of the reversed rule for every constructor which appears in the premises of one of the translated rules. A glance at the Bali rules does not reveal any reason for a non reversibility of them.

Method calls still remain a problem. The required annotations in the Bali syntax cannot be retrieved from explicit informations in the KeY syntax.

A further task will be to translate and proof further rules which have become able to translate, as the new version of Bali supports Java constructs such as labels, ... These new features could result in the necessity to use a different proof strategy.

For all proofs of the same type as the example, for every statement and expression appearing in the premises of the rule a reversed rule is needed. Therefore the more proofs have been done the more Bali rules for statements and expressions have been shown reversible, and can be reused for further proofs. So the more proofs have been constructed, the faster additional proofs are finished.

Appendix A

Running the proofs

This chapter contains the necessary information to run the proofs described in chapter 6.

A.1 Downloading Isabelle

First you need to install Isabelle/HOL. You can find the Isabelle homepage under http://isabelle.in.tum.de. The proofs shown here, are written for Isabelle2002/Isar. In order to install this, you need to download at least (for a Linux system):

- `Isabelle2002.tar.gz`
- `ProofGeneral.tar.gz`
- `polyml.base.tar.gz`
- `polyml_x86-linux.tar.gz`
- `HOL_x86-linux.tar.gz`

ProofGeneral is a frontend for XEmacs, which provides an interface to Isabelle. In order to run ProofGeneral, it is highly recommended to use a new version fo XEmacs rather than Emacs. Once you have downloaded these packages, you need to unpack them into a file like:

- `tar -C` {*path*} `Isabelle2002.tar.gz`
- `tar -C` {*path*} `ProofGeneral.tar.gz`
- `tar -C` {*path*} `polyml.base.tar.gz`
- `tar -C` {*path*} `polyml_x86-linux.tar.gz`
- `tar -C` {*path*} `HOL_x86-linux.tar.gz`

A.2 Installation of Bali

To be able to run any theories based on Bali, it is either possible to copy all files from the Bali directory {*path*}`/Isabelle/src/HOL/Bali` into the directory which contains the new theory files. Unfortunately this way all Bali theories with dependencies to the current theory have to be proved every time the proof is started. Building a Bali image avoids this. To build such an image change into to the {*path*}`/Isabelle/src/HOL` directory and run `isatool make HOL-Bali`. If this causes an error, change line 527 `@$(ISATOOL) usedir -g true $(OUT)/HOL Bali` into `@cd Bali; $(ISATOOL) usedir -b -g true $(OUT)/HOL Bali` and try it again. Building the Bali image takes a long time, don't be surprised if it lasts longer than one and a half hour (on a Pentium II processor). Once this is finished, you may invoke Isabelle with the additional parameter `-l HOL-Bali` or choose HOL-Bali in the ProofGeneral under Isabelle/Isar → Logics.

Building Bali and running Isabelle may require a change in the ML heap size. If you get strange error messages increase `ML_OPTIONS="-h 15000"` to `80000`.

A.3 Starting Isabelle

The change into the directory you used for installing the packages and type `Isabelle/bin/Isabelle &`. This starts Isabelle. Alternatively you may append the path to `Isabelle/bin` to your PATH variable.

In order to load a file, you can use the same shortcut as for Emacs: Ctrl-c Ctrl-f. Once you've loaded Final.thy (see the source code in the section below), you can start to rerun the proof. Here are a few shortcuts to help you:

- Ctrl-c Ctrl-n: run the next command of the proof
- Ctrl-c Ctrl-u: undo one step
- Ctrl-c Ctrl-Return: run the proof up to a certain point, specified by the position of the cursor.

A.4 Source code

You need to store the following source code in a file called Final.thy. The code here is adjusted to run under Isabelle2002 and the Bali version which belongs to it. However, at the time the research was done, this Bali version was not available. Therefore if you find a function or a datatype which is not explained in Chap. 2, it is most likely an adjustment to the new version.

```
theory Final = Eval:

(*** reverse rules: first instantiation of a cases rule, then use the
instantiation to prove a lemma which expresses the reverse rule directly ***)

(* produces an instanciation of cases rule *)
inductive_cases if_reverse: "(Norm s0, In1r (If(e) c1 Else c2), dummy_res, s2)
                              : eval G"

(* reverse if rule *)
lemma rev_if:"\<lbrakk>G\<turnstile>Norm s0 \<midarrow> If(cnd) a Else b
              \<rightarrow> s2\<rbrakk> \<Longrightarrow>
? v. ? s1. ((G\<turnstile>Norm s0 \<midarrow> cnd -\<succ> v \<rightarrow> s1)
            \<and> (G\<turnstile>s1\<midarrow>(if the_Bool v then a else b)
                   \<rightarrow> s2)) "
apply(erule if_reverse) (*produces two subgoals*)
   apply(simp_all) (*simplifies all subgoals*)
   apply(rule exI, blast)+ (* applies exI and blast to both subgoals *)
done


(* instanciation of split rule for composition of statements*)
inductive_cases comp_reverse: "(Norm s0, In1r (a;; b), dummy_res, s2): eval G"

(* reverse comp rule *)
lemma rev_comp:"G\<turnstile>Norm s0 \<midarrow> a;; b\<rightarrow> s2 ==>
? s1. ((G\<turnstile>Norm s0 \<midarrow> a \<rightarrow> s1) \<and>
       (G\<turnstile>s1 \<midarrow> b \<rightarrow> s2))"
apply(rule comp_reverse)
apply(simp_all)
(*2*)apply(rule exI)
     apply(blast)
```

```
done


(*short form of proof *)
lemma iftransformation: " ((G\<turnstile>Norm s0 \<midarrow>
    If(cnd) (Expr a) Else (Expr b)\<rightarrow> s1) ==>
    G\<turnstile> Norm s0 \<midarrow>Expr (cnd ? a : b) \<rightarrow> s1)"
apply(drule rev_if) (* work on assumption first *)
apply(safe)
apply(case_tac aa) (* start in an unknown state, case split to exception
                      or none- this is the only possibility to continue
                      working on the premisses, generates two subgoals *)
(*1*) apply(safe)
      apply(simp) (* generates two subgoals *)
        (* do this for the two subgoals *)
        apply(drule rev_expr, safe, rule Expr, rule Cond, assumption, simp)+

(*2*)apply(rule Expr) (* better than first simplifiying, this avoids case split *)
     apply(rule Cond) (* splits in two subgoals *)

     (*1*)apply(assumption) (*close one subgoal*)
     (*2*)apply(simp) (*removes two Isabelle ifs, splits in two subgoals*)

          apply(blast)+
done

end
```

Appendix B

Conversion table of symbols

$\Longrightarrow$	\ <Longrightarrow>
$\exists$	\ <exits> or ?
$\vdash$	\ <turnstile>
-	\ <midarrow>
$-\!\succ$	$-$\ <succ>
$\rightarrow$	\ <rightarrow>
$=\!\succ$	= \ <succ>
$\wedge$	\ <and>
$\neg$	~
$[\vert$	\ <lbrakk>
$\vert]$	\ <rbrakk>

Appendix C

Additional functions

Some KeYrules in Bali syntax required the specification of additional functions. The source code of them is listed here:

```
(* tests if two local variables have NOT the same name *)
not_in_lname :: "ename => lname => bool"

"not_in_lname e l == ? ln. if ( l = (EName ln)) then e ~= ln else True"


(* evaluates the value of a boolean expression *)
constdefs
ev_boolvar :: "expr => st => prog => val"
"ev_boolvar cnd s G == fst(SOME (v , s1). G\<turnstile> Norm s \<midarrow>
                                          cnd -\<succ> v \<rightarrow> s1)"



consts not_in_var ::      "ename => var => bool"
       not_in_expr ::     "ename => expr => bool"
       not_in_stmt ::     "ename => stmt => bool"
       not_in_exprlist :: "ename => expr list => bool"

primrec
"not_in_var nme (LVar ln)             = not_in_lname nme ln"
"not_in_var nme (FVar tn1 tn2 b e vn) = (nme ~= VNam vn)"
"not_in_var nme (AVar e1 e2)          = (not_in_expr nme e1 &
                                         not_in_expr nme e2)"


"not_in_expr nme (NewC  tn)                = True"
"not_in_expr nme (NewA  t e)               = True"
"not_in_expr nme (Cast  t e)               = not_in_expr nme e"
"not_in_expr nme (Inst  e rt)              = not_in_expr nme e"
"not_in_expr nme (Lit   vl)                = True"
"not_in_expr nme (Super)                   = True"
"not_in_expr nme (Acc   vr)                = not_in_var nme vr"
"not_in_expr nme (Ass   vr e)              = (not_in_var nme vr &
                                              not_in_expr nme e)"
"not_in_expr nme (Cond  e1 e2 e3)          = (not_in_expr nme e1 &
```

```
                                             not_in_expr nme e2 &
                                             not_in_expr nme e3)"
"not_in_expr nme (Call  rt1 rt2 invm e mn tylist exprlist)
                                          = not_in_exprlist nme exprlist"
"not_in_expr nme (Methd tn sg)            = True"
"not_in_expr nme (Body  tn smt)           = not_in_stmt nme smt"


"not_in_stmt nme Skip                = True"
"not_in_stmt nme (Expr  e)           = not_in_expr nme e"
"not_in_stmt nme (Comp  s1 s2)       = (not_in_stmt nme s1 &
                                        not_in_stmt nme s2)"
"not_in_stmt nme (If_   e s1 s2)     = (not_in_expr nme e & not_in_stmt nme s1                                        & not_in_stmt nme s2)"
"not_in_stmt nme (Loop l e s)        = (not_in_expr nme e & not_in_stmt nme s)"
"not_in_stmt nme (Throw e)           = not_in_expr nme e"
"not_in_stmt nme (TryC  s1 tn vn s2) = (not_in_stmt nme s1 & not_in_stmt nme s2                                      & (nme ~= VNam vn))"
"not_in_stmt nme (Fin   s1 s2)       = (not_in_stmt nme s1 &
                                        not_in_stmt nme s2)"
"not_in_stmt nme (Init  tn)          = True"

"not_in_exprlist nme []     = True"
"not_in_exprlist nme (e#el) = (not_in_expr nme e & not_in_exprlist nme el)"
```

Bibliography

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.

[2] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integration of automated and interactive theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications*, volume II, chapter 4, pages 97–116. Kluwer, 1998.

[3] Bernhard Beckert. A dynamic logic for Java Card. In *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France*, 2000. Available at i12www.ira.uka.de/~key.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[6] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation*. IOS Press, 2000. To appear.

[7] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[8] Rational Software Corp. et al. *Object Constraint Language Specification, version 1.1*, 1997.

[9] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.1 Application Programming Interfaces, Draft 2, Release 1.3*, 1998.

[10] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Applet Developer's Guide*, 1998.

[11] David von Oheimb. Axiomatic semantics for Java^{light}. In Sophia Drossopoulou, Susan Eisenbach, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Proceedings, Formal Techniques for Java Programs, Workshop at ECOOP'00, Cannes, France*, 2000.

[12] David von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, Januar 2001.