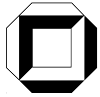

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Verification of Java Floating-Point Arithmetic

Fakultät für Informatik
Universität Karlsruhe (TH)

Diplomarbeit
von

cand. inform. Denis Lohner

Juni 2008

Betreuer: Dipl. Inform. Benjamin Weiß
Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt

Institut für Theoretische Informatik - KeY Project

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine weiteren als die angegebenen Hilfsmittel verwendet zu haben.

Karlsruhe, den 3. Juni 2008

Denis Lohner

Deutsche Zusammenfassung

Um größt mögliches Vertrauen in Software zu erlangen, werden im Softwareentwicklungsprozess heutzutage viele Ressourcen für Testzwecke ausgegeben. Es gibt jedoch auch die Möglichkeit mithilfe von formaler Spezifikation und Verifikation die Sicherheit und Robustheit von Software zu erhöhen. Mit dem KeY-Tool [30] wird hierzu ein Werkzeug angeboten, dass als Grundlage für die Integration formaler Methoden in den Softwareentwicklungsprozess bildet.

Allerdings bot das Tool bisher keine Unterstützung für Gleitkomma-Arithmetik. Diese wird jedoch von vielen sicherheitskritischen Applikationen verwendet. Als Beispiel sei hier nur der Erststart der Trägerrakete „Ariane 5“ erwähnt, welcher aufgrund eines Softwarefehlers mißlang, bei dem Gleitkomma-Arithmetik eine wesentliche Rolle spielte.

Diese Arbeit befasst sich daher zunächst mit der Gleitkomma-Arithmetik als solche, wobei eine mathematische Beschreibung einer generalisierten (exakten) Form der Gleitkomma-Arithmetik hergeleitet wird. Weiter werden die Eigenheiten der Gleitkomma-Arithmetik, wie sie in der Programmiersprache Java zum Einsatz kommt, herausgearbeitet, um schließlich auf die dadurch für die formale Behandlung entstehenden Probleme einzugehen. Dabei stellt sich heraus, dass insbesondere die Einführung der Werte `NaN` (*Not-a-Number*) und der vorzeichenbehafteten Nullen sowie die Tatsache, dass eine Gleitkomma-Operation im Allgemeinen nicht exakt ist und daher gerundet werden muss, den Entwurf einer Logik erschweren.

Im nächsten Schritt wird dann eine mathematische Beschreibung der Rundungsfunktion hergeleitet, welche sich ohne größere Probleme in einer Logik erster Stufe formalisieren läßt. Schließlich wird die dynamische Logik JavaCardDL$^+$ sowie ein Kalkül dafür entworfen, welche eine Erweiterung der Logik und des Kalküls für JavaCard aus [6] sind und die nötigen Voraussetzungen für die formale Behandlung von Java Gleitkomma-Arithmetik beinhalten. Dabei ist darauf zu achten, dass durch die Erweiterung der Typhierarchie der Logik manche Regeln des Kalküls für JavaCard nicht mehr korrekt sind und daher ersetzt werden müssen.

Contents

1 Preface **7**

1.1 Motivation 7

1.2 The KeY project 7

1.2.1 The KeY tool 8

1.3 Outline 8

2 Prerequisites **9**

2.1 The IEEE 754 floating-point formats 9

2.2 Mathematical observations 9

2.2.1 Defining a mathematical structure for floating-point arithmetic 10

2.2.2 Improving the definition 11

2.2.3 Properties of the defined structure 13

3 Floating point types and operations in Java **15**

3.1 History 15

3.2 Differences between Java and IEEE 754 16

3.2.1 Value representations 16

3.2.2 Rounding and Conversions 18

3.2.3 Exceptions, Flags and Traps 20

3.2.4 Operations 21

3.2.5 The modifier strictfp 22

4 Towards verification of floating point arithmetic **23**

4.1 Verification Goal 23

4.2 Semantics of Java's `+`, `-`, `*` and `/` 25

4.3 Signed Zeros 26

4.4 NaNs 27

4.5 Rounding issues 28

4.6 Platform dependency issues 29

4.7 Conclusion 29

5 Rounding **31**

5.1 Related Work 31

5.2 Modelling rounding by direct computation 31
5.3 Modelling rounding by relations 32
5.4 Rounding up and down 32
5.4.1 Unbounded rounding 32
5.4.2 Bounded rounding 34

6 The logic part 37
6.1 Extensions to JavaCardDL 37
6.1.1 Type hierarchy 37
6.1.2 Signature 38
6.1.3 Syntax 42
6.1.4 Semantics 43
6.2 A calculus for JavaCardDL+ 43
6.2.1 A new assignment rule 44
6.2.2 Program transformation and simplification rules 45
6.2.3 Rules for Java semantics 47
6.2.4 Rules for algorithmic semantics 49
6.2.5 Rules for Handling exact floating-point operations 50

7 Conclusion 55
7.1 Future work 55

A Semantics of predefined rigid symbols 57
A.1 Symbols having the same semantics in all calculuses 57
A.2 Semantics reflecting Java's behavior 60
A.3 Semantics for algorithmic verification 62

Bibliography 63

1 Preface

1.1 Motivation

There is no larger software project without bugs in the code. Therefore, time- and resource-consuming tests are scheduled within the software development process. However, not all bugs can be found by testing in general. Especially for security-critical applications this can have serious consequences. For example, the destruction of the rocket "Ariane 5" in 1996 was the result of a software bug. In particular, the conversion of a floating-point number to an integer caused an overflow in the software, that was not caught. As an effect of this, some status data was (spuriously) interpreted as actual flight attitude and the thrusters were pulled to the edge. The rocket was just about to break apart when it self-destructed.

The use of formal methods is one approach to gain more trust in a specific piece of software. Thus, analyzing, specifying and verifying software increases the confidence in the software code. There are different approaches of integrating formal methods in the software development process. One of them is the KeY project (see below). However, many security-critical applications use floating-point arithmetic (especially if physics are involved) but the KeY tool only supports integer arithmetic by now. Thus, this thesis is the first step to integrate floating-point arithmetic within the KeY project. The JAVA floating-point arithmetic will be analyzed, a formalization of a rounding function is derived and the logic used by the KeY tool, JAVACARDDL, will be extended such that basic floating-point operations can be analyzed within the logic.

1.2 The KeY project

In 1998 the KeY project [30] was started at the University of Karlsruhe. It aims at the integration of formal methods into the industrial software development process. Therefore, a tool – the KeY tool – has been developed which consists of an interactive theorem prover and also serves as a base for further applications such as an Eclipse [2] plugin or a symbolic debugger [5]. The programming language, the KeY project

originally aimed at, is JavaCard which is a Java variant for smart card applications. However, until now many features of Java that are not part of JavaCard can be handled by the tool and there is also a variant for the language C [12].

1.2.1 The KeY tool

The tool itself is an interactive theorem prover based on a dynamic first order logic (JavaCardDL) and uses a sequent calculus, that also supports term rewrite rules. The user may – at any time – apply rules by hand or lets choose the rules by one of the built-in strategies. The strategies provide a high level of automation so that only a few proof steps must be done by hand. Therefore, the proof environment supports interactive proving steps by easy-to-use operations.

1.3 Outline

This thesis is structured as followed. Chapter 2 will give a short introduction into the IEEE 754 standard [20] and derives a mathematical description of floating-point arithmetic. Chapter 3 then analyzes the Java floating point arithmetic and will show up differences and similarities with the IEEE 754 standard. In chapter 4 some issues relevant for verification purpose will be discussed, while chapter 5 introduces a formalization of a rounding function. Finally, chapter 6 presents the extensions made to JavaCardDL that allow floating-point arithmetic to be handled by the logic. The chapter also presents a calculus for the derived logic.

2 Prerequisites

2.1 The IEEE 754 floating-point formats

The most common standard for modern computer arithmetic is the "IEEE Standard for Binary Floating-Point Arithmetic" (ANSI/IEEE 754-1985, [20]). In the scope of this thesis this standard is referred to as *IEEE 754*. The standard defines four different formats in two groups, the *basic* formats and the *extended* formats. The two basic formats *Single* and *Double* are specified very precisely whereas the extended formats *Single extended* and *Double extended* let some choice to the implementer within given constraints. Further, the IEEE 754 standard also defines four different rounding modes as mappings from real numbers to a floating-point format. These cover *round to nearest*, *round up*, *round down* and *round to zero*.

Also, the four basic arithmetic operations *addition*, *subtraction*, *multiplication* and *division* as well as a *remainder operator* and extracting the *square root* are part of the standard (section 5, [20]) and are defined very precisely. Thus, every implementation of the standard should behave exactly the same way while computing.

As neither real arithmetic nor floating-point arithmetic is closed under the operations defined in IEEE 754, situations can occur, where the result of an operation is mathematically not defined (such as division by zero) or does not fit in the destination's format. For such cases, the standard (section 7, [20]) defines five exceptions: *Invalid Operation*, *Division by Zero*, *Overflow*, *Underflow* and *Inexact*. In each case, the implementer may then return the value specified by the standard or implement a trap handler (see section 8, [20]) to deal with the exception.

More details on the IEEE 754 standard will be given when they are needed in this thesis.

2.2 Mathematical observations

In this section a mathematical description of floating-point arithmetic will be introduced. Some properties of this description will also be emphasized. As floating-point

arithmetic is an approximation to real arithmetic it is important to have closely related mathematical structures. However, computations in floating-point arithmetic are done with bits and bytes which means that the actual implementation is more closely related to integer arithmetic. Therefore, a description of floating-point arithmetic is used which can be embedded into real arithmetic but is solely based on integer arithmetic.

2.2.1 Defining a mathematical structure for floating-point arithmetic

The idea is to represent every floating-point number by two unbounded integers – the mantissa and the exponent. Thus, a floating-point number f is a tuple $(m, e) \in \mathbb{Z}^2$ such that f is associated with the real number $m \cdot 2^e$. The set of all tuples (m, e) is denoted by $\mathbb{F}$.

Definition 2.1
Let $\mathbb{F}$ be the set of all tuples (m, e) with $m,\ e \in \mathbb{Z}$.

$$\mathbb{F} := \{(m, e) \in \mathbb{Z}^2\}$$

*The **value** of an element $f = (m, e) \in \mathbb{F}$ is the real value of the expression $m \cdot 2^e$.*

Remark: *There are different elements in $\mathbb{F}$ which have the same value. For example $(2, 0)$ and $(1, 1)$ both have the value two.*

Next, the basic arithmetic operations are defined.

Definition 2.2
The relation $\leq : \mathbb{F} \times \mathbb{F}$ is defined by

$$(m_1, e_1) \leq (m_2, e_2) \quad :\Longleftrightarrow \quad m_1 \cdot 2^{e_1} \leq m_2 \cdot 2^{e_2}$$

The arithmetic operations addition $+ : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, multiplication $\cdot : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ and unary minus $- : \mathbb{F} \to \mathbb{F}$ are defined by

$$\begin{aligned}
(m_1, e_1) + (m_2, e_2) &:= (m_1 \cdot 2^{e_1 - \min(e_1, e_2)} + m_2 \cdot 2^{e_2 - \min(e_1, e_2)}, \min(e_1, e_2)) \\
(m_1, e_1) \cdot (m_2, e_2) &:= (m_1 \cdot m_2, e_1 + e_2) \\
-(m_1, e_1) &:= (-m_1, e_1)
\end{aligned}$$

Note that these operations are well-defined as for the addition both powers of two are greater than one (the exponents of $2^{e_1 - \min(e_1, e_2)}$ and $2^{e_2 - \min(e_1, e_2)}$ are both non-negative).

Lemma 2.3
For every $f, f_1, f_2 \in \mathbb{F}$ and $r, r_1, r_2 \in \mathbb{R}$ it holds, that if r is the value of f and r_i is the value of f_i, $i \in \{1, 2\}$,

- *if $f_1 \leq f_2$ then $r_1 \leq r_2$,*
- *if $f = f_1 + f_2$ then $r = r_1 + r_2$,*
- *if $f = f_1 \cdot f_2$ then $r = r_1 \cdot r_2$ and*
- *if $f_1 = -f_2$ then $r_1 = -r_2$.*

Proof. The proof of this lemma can be done by just applying the definitions 2.1 and 2.2. □

2.2.2 Improving the definition

Studying the algebraical structure of the magma $(\mathbb{F}, +)$, it is obvious that $(\mathbb{F}, +)$ is not even a monoid as there is no unique zero. Therefore, the set $\mathbb{F}$ is not appropriate to serve as a mathematical description of floating-point arithmetic. However, it is possible to define an equivalence relation over $\mathbb{F}$ such that the resulting quotient set has a well defined structure similar to $\mathbb{Z}$ (but – in contrast to $\mathbb{Z}$ – is dense in $\mathbb{R}$).

Definition 2.4
Let $f_1, f_2 \in \mathbb{F}$ and $r_1, r_2 \in \mathbb{R}$ such that the value of f_1 is r_1 and the value of f_2 is r_2. The relation $\sim: \mathbb{F} \times \mathbb{F}$ is defined by

$$f_1 \sim f_2 \quad :\Longleftrightarrow \quad r_1 = r_2$$

Remark: *As "$=$" is an equivalence relation over $\mathbb{R}$, "$\sim$" is also an equivalence relation (over $\mathbb{F}$).*

Thus, we can define the notion of equivalence class and quotient set.

Definition 2.5
Let $a \in \mathbb{F}$

- *The **equivalence class** $\widehat{a}$ is defined as usual by $\widehat{a} := \{b \in \mathbb{F} : a \sim b\}$.*
- *The set $\widehat{\mathbb{F}} := \mathbb{F}/\sim = \{\widehat{a} : a \in \mathbb{F}\}$ is the quotient set of $\mathbb{F}$ by $\sim$.*

There are two important properties of these equivalence classes which are outlined in the following lemma.

Lemma 2.6
The following statements are true.

1. *Let $a \in \mathbb{F}$ and $v_a \in \mathbb{R}$ be the value of a, then every $b \in \widehat{a}$ has the value v_a.*
2. *If $a \in \mathbb{F}$ and $b \in \mathbb{F}$ have the same value, then $\widehat{a} = \widehat{b}$.*

Proof. These properties follow directly from the fact that all $f \in \mathbb{F}$ are defined equivalent by $\sim$, if and only if they have the same value (see definition 2.4). ☐

Remark: *Because of these properties, the* ***value of an equivalence class*** *can be said to be the value of its members.*

Now, the definition of the above operations and relations must be applied to equivalence classes.

Definition 2.7
Let $f_1, f_2 \in \mathbb{F}$. The relation $\leq: \widehat{\mathbb{F}} \times \widehat{\mathbb{F}}$ and operations $+ : \widehat{\mathbb{F}} \times \widehat{\mathbb{F}} \to \widehat{\mathbb{F}}$, $\cdot : \widehat{\mathbb{F}} \times \widehat{\mathbb{F}} \to \widehat{\mathbb{F}}$ and $- : \widehat{\mathbb{F}} \to \widehat{\mathbb{F}}$ are defined by:

$$\begin{aligned} \widehat{f_1} \leq \widehat{f_2} \quad &:\Longleftrightarrow \quad f_1 \leq f_2 \\ \widehat{f_1} + \widehat{f_2} \quad &:= \quad \widehat{f_1 + f_2} \\ \widehat{f_1} \cdot \widehat{f_2} \quad &:= \quad \widehat{f_1 \cdot f_2} \\ -\widehat{f_1} \quad &:= \quad \widehat{-f_1} \end{aligned}$$

Lemma 2.8
Definition 2.7 implies that,

1. *the operations on equivalence classes are well-defined and*
2. *lemma 2.3 also applies if f, f_1 and f_2 are equivalence classes ($\in \widehat{\mathbb{F}}$).*

Proof. Let $a, b \in \mathbb{F}$ and $a' \in \widehat{a}$ and $b' \in \widehat{b}$. Further let $v_a \in \mathbb{R}$ be the value of a and $v_b \in \mathbb{R}$ be the value of b.

1. As $a \sim a'$ and $b \sim b'$ the value of a' is v_a and the value of b' is v_b. Therefore, according to lemma 2.3, the value of $a + b$ is $v_a + v_b$ which is also the value of $a' + b'$. Thus, $a + b \sim a' + b'$ and therefore $\widehat{a + b} = \widehat{a' + b'}$.

   The same applies to multiplication and slightly adapted to unary minus.
2.
   - If $\widehat{a} \leq \widehat{b}$ then by definition $a \leq b$ and according to lemma 2.3 $v_a \leq v_b$.
   - Let $c \in \mathbb{F}$ with $\widehat{c} = \widehat{a} + \widehat{b}$ then by definition $\widehat{c} = \widehat{a + b}$ and thus $c \sim a + b$. Therefore, the value of c (and of $\widehat{c}$) is the value of $a + b$ (and of $\widehat{a} + \widehat{b}$).

- Multiplication and unary minus are treated analogously.

□

2.2.3 Properties of the defined structure

Studying the algebraic structure of $\widehat{\mathbb{F}}$ one obtains the following theorem.

Theorem 2.9
$(\widehat{\mathbb{F}}, +, \cdot)$ is a commutative ring with unit $\widehat{e} := \widehat{(1,0)}$ and zero $\widehat{o} := \widehat{(0,0)}$ (but not a field).

Proof. Let $a, b, c \in \widehat{\mathbb{F}}$ and $v_a, v_b, v_c \in \mathbb{R}$ be the values of a, b and c.

1. $(\widehat{\mathbb{F}}, +)$ is an abelian group.
   - According to lemma 2.6, the value of $(a+b)+c$ is $(v_a+v_b)+v_c = v_a+(v_b+v_c)$ which is the value of $a+(b+c)$. Thus, as $(a+b)+c$ has the same value as $a+(b+c)$, they are the same element of $\widehat{\mathbb{F}}$ (lemma 2.6). Hence, the addition is associative.
   - As the value of $a+b$ is the same as of $b+a$, the addition is commutative.
   - The element $o = \widehat{(0,0)}$ is unique zero in $\widehat{\mathbb{F}}$, since $a+o$ has value v_a (the value of o is zero) and thus $a+o = a$.
   - The inverse of a is $-a$, since the value of $a+(-a)$ is $v_a + (-v_a) = 0$ and therefore $a + (-a) = o$.
2. $(\widehat{\mathbb{F}}, \cdot)$ is a commutative monoid with the same justification as above (replace $+$ by $\cdot$, $o = \widehat{(0,0)}$ by $e = \widehat{(1,0)}$ and omit the inverse).
3. Multiplication distributes over addition: The value of $a \cdot (b+c)$ is $v_a \cdot (v_b + v_c) = (v_a \cdot v_b) + (v_a \cdot v_c)$. Thus, by lemma 2.6, $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$. As multiplication is commutative, $(a+b) \cdot c$ equals $(a \cdot c) + (b \cdot c)$.
4. $(\widehat{\mathbb{F}}, +, \cdot)$ is not a field, as for example $\widehat{(3,0)}$ has no multiplicative inverse.

□

Further to relate $\widehat{\mathbb{F}}$ closer to $\mathbb{R}$, an embedding of $\widehat{\mathbb{F}}$ into $\mathbb{R}$ is defined by the following theorem.

Theorem 2.10
Let $\sigma : \widehat{\mathbb{F}} \to \mathbb{R}$ *be defined such that* $\sigma(a)$ *is the value of* a *which means*

$$\sigma(\widehat{(m,e)}) := m \cdot 2^e$$

Then σ *is an injective ring homomorphism that preserves the ordering relation* $\leq$ *(i.e. for* $a, b \in \widehat{\mathbb{F}}$*,* $a \leq b \Leftrightarrow \sigma(a) \leq \sigma(b)$*).*

Proof. The proof follows from lemma 2.8.

- It follows directly from lemma 2.8, that $\sigma(a+b) = \sigma(a) + \sigma(b)$ as well as $\sigma(a \cdot b) = \sigma(a) \cdot \sigma(b)$. Further, $\sigma(\widehat{1}) = \sigma(\widehat{(1,0)}) = 1 \cdot 2^0 = 1$. Thus, σ is a ring homomorphism.
- To show σ is injective, it is sufficient to show, that $\ker(\sigma) = \{\widehat{0}\}$.
  Let $a \in \ker(\sigma) \subseteq \widehat{\mathbb{F}}$ such that $a = \widehat{(m,e)}$. Thus, $\sigma(\widehat{(m,e)}) = 0$ and therefore, $m \cdot 2^e = 0$. As 2^e is always non-zero, m must equal zero and hence $a = \widehat{(0,e)} = \widehat{0}$. Consequently, $\widehat{0}$ is the only element in the kernel of σ.
- The homomorphism σ preserves the order relation. Let $a, b \in \widehat{\mathbb{F}}$ such that $a = \widehat{(m_1, e_1)}$ and $b = \widehat{(m_2, e_2)}$. Then, $a \leq b \overset{Def.\ 2.7}{\Leftrightarrow} (m_1, e_1) \leq (m_2, e_2) \overset{Def.\ 2.2}{\Leftrightarrow} m_1 \cdot 2^{e_1} \leq m_2 \cdot 2^{e_2} \Leftrightarrow \sigma(a) \leq \sigma(b)$.

□

Thus, the ordered ring $(\widehat{\mathbb{F}}, +, \cdot, \leq)$ is isomorphic to $(\sigma(\widehat{\mathbb{F}}) \subseteq \mathbb{R}, +, \cdot, \leq)$. Therefore, every element $a \in \widehat{\mathbb{F}}$ can be seen as the real number $\sigma(a)$ and we consider $\widehat{\mathbb{F}}$ to be a subset of $\mathbb{R}$.

Remark: *The subset* $\widehat{\mathbb{F}} \subseteq \mathbb{R}$ *is dense in* $\mathbb{R}$*, which means any real number can be approximated arbitrarily close in* $\widehat{\mathbb{F}}$*.*

A floating point format can now be seen as a finite subset of $\widehat{\mathbb{F}}$ enriched with some special values demanded by the IEEE 754 standard such as NaN.

3 Floating point types and operations in Java

3.1 History

When Java was developed in the mid 1990s two floating point types `float` and `double` were introduced with the Java *Language Specification* [16], corresponding to the 32-bit single-precision and 64-bit double-precision IEEE 754 basic formats as specified in the ANSI/IEEE Standard 754-1985 [20]. However, this standard defines two more formats using a wider exponent range as well as higher precision but gives only minimal requirements for implementing those extended formats. That means, hardware or software implementers can offer these formats differently. Thus, as Java was designed to be platform independent, in Java *vers. 1.0* all floating point operations were restricted to the two basic formats.

However, as Java got more and more popular and, by the same time, high precision hardware floating point units got more common (Intel^{TM} integrated the 80387 floating point unit with 80-bit precision into their i386-processors), the growing Java community applied for an integration of higher precision floating point numbers into the Java Language Specification (JLS) [10]. As a result of that, the JLS (second and third edition) [14, 15] now defines additional, so called "extended-exponent value sets" for both the `float` and `double` primitive type, that allow direct use of the Intel^{TM} x87 floating point unit. However, there are no additional types for these value sets. It is just, that the specification allows to compute intermediate results of floating point operations with greater exponent range. In return, a new modifier `strictfp` (see 3.2.5) has been introduced that allows to force floating point operations being computed within the basic formats only and therefore being deterministic. This approach however, was not preferred by everyone [21].

Hence, the today's primitive Java types `float` and `double` conform neither to the IEEE 754 basic formats nor to the IEEE 754 extended formats but rather are a mixture of both. Fortunately, there is the `strictfp` modifier that at least ensures conformance[1]

[1] But just conformance concerning the representation of real values, not concerning the operations as we will see in section 3.2.

to the IEEE 754 standard.

3.2 Differences between Java and IEEE 754

As pointed out in the last section, JAVA fails to conform to the IEEE 754 standard. However, a large part of the JAVA floating point specifications can be seen as a subset of the IEEE 754 specifications. This section will give a detailed overview of the differences and similarities.

3.2.1 Value representations

In general, real values can be expressed in the form

$$(-1)^s \cdot (1+f) \cdot 2^{exp}$$

where s (the *sign*) is either 0 or 1, f (the *fraction-part*) is a non-negative real number smaller than 1 and exp (the *exponent*) is an arbitrary integer. As $1 \leq (1+f) < 2$ always holds, every non-zero real number has exactly one such representation. Thus, the set of real numbers can be expressed by

$$\mathbb{R} = \{(s, f, exp) |\ s = 0 \vee s = 1,\ 0 \leq f < 1,\ exp \in \mathbb{Z}\} \cup \{0\}$$

A specific binary floating point format is generally specified by encoding the fraction-part f and the exponent exp in binary. The fraction-part is thereby restricted to a certain precision and the exponent range is limited by a lower bound E_{min} and an upper bound E_{max}. Additionally there are some special representations for *signed zeros*, *signed infinities* and *NaN* (Not-a-Number) values.

A number representable by a triple (s, f, exp) where f and exp correspond to the restrictions of a given format are called *normalized*.

3.2.1.1 Gradual Underflow

As the smallest positive normalized number in a specific format is $+1 \cdot (1+0) \cdot 2^{E_{min}} = 2^{E_{min}}$, the IEEE 754 standard as well as the JAVA Language specification also define *denormalized* numbers, that can be expressed as $(-1)^s \cdot (0+f) \cdot 2^{E_{min}}$. This approach helps to fill the gap between 0 and $\pm 2^{E_{min}}$ and is referred to as *gradual underflow*.

Parameter	single	single-extended	double	double-extended
Precision of fraction-part (in bits)	23	≥ 31	52	≥ 63
Width of the exponent (in bits)	8	≥ 11	11	≥ 15
E_{max}	+127	$\geq +1023$	+1023	$\geq +16383$
E_{min}	−126	≤ -1022	−1022	≤ -16382

Table 3.1. IEEE floating point formats

3.2.1.2 IEEE 754 formats

The IEEE 754 standard defines four different floating point formats (**single, double, single-extended** and **double-extended**) for which an implementation of the standard must at least provide the single format. The single and double format are both specified rather precise, whereas the extended formats leave some choice to the implementer. Concrete parameters for the formats are given in table 3.1.

There are also two reserved exponent values defined for encoding denormalized values and signed zeros ($exp = E_{min} - 1$) as well as infinities and NaNs ($exp = E_{max} + 1$).

3.2.1.3 Java formats

According to the Java Language Specification (3rd edition) [15] floating point elements in Java can be declared either as type `float` (32-bit precision) or as type `double` (64-bit precision). However, the representation of the numbers can originate from different value sets. Every Java implemention has to provide the *float value set* and the *double value set* that correspond to the IEEE 754 single precision and double precision formats. But there may also be a *single-extended-exponent value set* and a *double-extended-exponent value set* that require to have the same precision in the fraction-part as the standard value sets but allow the exponent range to be much wider (see table 3.2 for detailed parameters). Note, that the float-extended-exponent value set (double-extended-exponent value set) do **not** conform to the IEEE 754 single-extended (double-extended) format. Further, the extended-exponent value sets are not visible to a Java programmer, they are just there to store intermediate results which otherwise would result in an over- or underflow.

Parameter	float	float-extended	double	double-extended
Precision of fraction-part (in bits)	23	23	52	52
Width of the exponent (in bits)	8	≥ 11	11	≥ 15
E_{max}	+127	$\geq +1023$	+1023	$\geq +16383$
E_{min}	−126	≤ -1022	−1022	≤ -16382

Table 3.2. JAVA floating point value set parameters

3.2.2 Rounding and Conversions

3.2.2.1 Rounding

IEEE 754

Whenever the infinitely precise result of a computation does not fit in the required destination format, rounding must be applied. Therefore, according to the IEEE 754 standard, four different modes for rounding must be supported between which a user may choose.

- **round-to-nearest**: This is the default rounding mode. The result is the value representable in the destination format nearest to the infinitely precise result. If there are two such values, the one with the least significant bit 0 in its fraction-part is chosen. If the infinitely precise result exceeds $2^{E_{max}}(2 - 2^{-(p+1)})$ (where p is the precision of the fraction part) in magnitude, the result is an infinity with the appropriate sign.
- **round-to-zero**: The result is the value representable in the destination format closest to the infinitely precise result that is not greater in magnitude.
- **round-to-positive-infinity**: The result is the value representable in the destination format closest to the infinitely precise result, but not smaller than it.
- **round-to-negative-infinity**: The result is the value representable in the destination format closest to the infinitely precise result, but not greater than it.

Java

In the JAVA world the user has no choice than to accept the IEEE default rounding mode (which is round-to-nearest) for nearly any floating point computation. The sole exception is that when casting a floating point value to an integral type the round-to-zero mode is used.

3.2.2.2 Converting between floating point types

Converting (casting) between floating point types in Java is done as demanded by the IEEE 754 standard except that always round-to-nearest is used in the narrowing conversion. The widening conversion is exact whenever only standard value sets are used.

3.2.2.3 Converting from integral types

Converting from `byte`, `char` or `short` to `float` or `double` is always exact as well as conversion from `int` to `double`. As the precision of the fraction-part of `float` is smaller than the precision of `int` or `long`, the conversion from `int` (or `long`) to `float` may lead to a less precise result, but in a correctly rounded (round-to-nearest) one. The same applies for the conversion from `long` to `double`. This is also conforming to IEEE 754.

3.2.2.4 Converting to integral types

Due to missing floating point exceptions (see section 3.2.3) in Java, conversion from floating point types to integral types also does **not** conform to IEEE 754.

IEEE 754
The IEEE 754 standard demands that converting from special values (NaNs and infinities) results in an invalid operation.

Java
However, the "Java conversion algorithm" as described in §5.1.3 of the Java Language Specification (3rd edition) [15] converts a NaN value to zero. In all other cases, the floating point number is first converted to `long` (if `long` is the destination type) or `int` (if any other integral type is the destination type) by applying rounding in the round-to-zero mode and is then converted to the desired type by the integer conversion rules. Also, if the floating point number is too large in magnitude (even if it is ±infinity) to fit into type `long` (or `int`), it is (in the first step) converted to the largest number in magnitude of type `long` (or `int`) with the same sign as the floating point number.

So for example, if the number `3.05463295817e8` (represented as `double`) is converted to type `int`, the result will be the correctly rounded (round-to-zero) number `305463295`. However, if the same floating point number is converted to type `short`,

the result will be `-1`. Since the conversion is done in two steps as described above, the number is first converted to `int`, and then converted to `short` by truncating the upper 16 bits of the binary representation of `305463295`, which is (in hexadecimal form) `1234FFFF`. So, the result of the conversion to `short` will be `FFFF` in hexadecimal form, which is `-1` in decimal form.

3.2.2.5 Binary ↔ Decimal conversion

IEEE 754

According to the IEEE 754 standard, conversion of decimal strings to single (double) precision floating point numbers shall be provided for decimal strings that have values up to $9.99999999 \cdot 10^{107}$ ($9.9999999999999999 \cdot 10^{1015}$) in magnitude as well as down to 10^{-99} (10^{-999}) and have a precision of up to nine (17) digits in the significand. For the conversion to decimal strings the standard specifies that every single and double precision floating point number must have a decimal string representation such that, converted back to a floating point number of the same type, results in the identity as long as the decimal representation has a nine digit precision for the single format or 17 digit precision for the double format.

Java

In Java for both types `float` and `double` it holds that every decimal representation of a floating point number can be used as a floating point literal, as long as there is a normalized or denormalized number in the corresponding type, that represents the correctly rounded (using round-to-nearest) version of the given number. That means, numbers equal or greater in magnitude than $2^{127}(2-2^{24}) = 2^{128} - 2^{103}$ and numbers equal or smaller in magnitude than 2^{-150} (except `+0.0` and `-0.0`) can not be used as literals for the type `float`. For `double` the bounds are $2^{1024} - 2^{970}$ and 2^{-1075}.

As the limits of decimal to binary conversion in Java do not exceed the limits specified in the IEEE standard, Java is also not conforming to the standard in this point. But as there is a different decimal string representation for every floating point value (except infinities and NaN), Java fulfills all requirements of the standard concerning conversion between binary and decimal except the greater range.

3.2.3 Exceptions, Flags and Traps

The IEEE 754 standard also specifies that there must be some type of feedback to the user about a computation's result, such as if the result is exact or if it is rounded.

Therefore the concept of exceptions is used. The standard claims the presence of at least five exception types. Roughly summarized these exceptions are the following.

- **Invalid Operation**: This exceptions occurs for operations that were forbidden in real arithmetics such as computing the square root of a negative number, and also for operations involving NaNs and infinities if there is no reasonable result such as multiplying 0 and infinity.
- **Division by Zero**: This exception occurs if the divisor of a division is zero and the dividend is a normalized or denormalized non-zero number (Note, that division of zero by zero yields an invalid operation exception).
- **Overflow**: Overflow occurs whenever the number that would be a computations exactly rounded result (if the exponent range were unbounded) exceeds the destination formats largest number in magnitude.
- **Underflow**: The underflow exception occurs if a computations exact (non-zero) result is so tiny in magnitude that it is rounded to a denormalized number or zero in the destinations format.
- **Inexact**: This exception occurs if the rounded result of an operation is not equal to the exact result.

For each exception the standard specifies that there must be a status flag accessible to the user. Also the user may specify a trap handler for each exception that is called whenever the corresponding exception occurs.

As Java neither provides those status flags nor lets the user declare the trap handlers, there is absolutely no support of floating point exceptions in Java. Note that the assumption a resulting infinity is always attributed to an overflow is not right. Division by zero may also lead to an infinity. So the only way to check whether an infinity results from overflow or division by zero is to check the appropriate status flags, which is impossible in Java.

3.2.4 Operations

All mathematical operations required by the IEEE 754 standard are supported in Java. These are addition, subtraction, multiplication and division as well as extracting the square root and finding the remainder. However, Java's remainder operator `%` does not conform to the specification in the IEEE standard. There is a special method `IEEEremainder` in the class `java.lang.Math` to obtain the remainder as specified by the standard, but this method exists only for arguments of type `double`. This however, is no drawback as the computation of the remainder is always exact and is not affected by rounding. Therefore, the `double` precision result of a remainder computation of two

`floats` can be converted to `float` without loss of precision. Thus, the `IEEEremainder` method can be used to compute the single precision remainder of two `floats`.

The results of all other mathematical operations are those conforming to the IEEE standard as long as only standard value sets (see section 3.2.1.3) are used and round-to-nearest is the rounding mode. An invalid operation (such as "0/0") always results in a NaN, whereas overflow and division by zero results in $\pm\infty$ and underflow results in ± 0.

Some interesting and non-intuitive examples for floating point computations are given in chapter 4.

3.2.5 The modifier strictfp

To overcome the problem of over- and underflow within intermediate results of a computation, the extended-exponent value sets (see 3.2.1.3) had been introduced with the second version of the Java Language specification [14]. The price was loss of determinism in floating point computation results, as nearly all floating point operations became dependent on the virtual machine's implementation and even on the hardware implementation of floating point operations.

Thus, as Java was at first intentionally designed to be platform independent and deterministic, a new modifier keyword `strictfp` had been introduced to restrict floating point computations and their intermediate results to the standard value sets only.

The `strictfp` modifier can be attached to class-, interface- or method-declarations and affects all floating point expressions within the declaration's scope. If a non-`strictfp` method is called from such an expression and the return type is either `float` or `double`, then the result must be mapped to the corresponding standard value set according to §5.1.13 "Value Set Conversion" in [15].

As value set conversion in non-`strictfp` expressions is nondeterministic among different implementations of Java, the remainder of this work will only consider `strictfp` floating point expressions.

Unfortunately, a very common Java compiler, namely the GNU Java compiler *gcj*, does not support this modifier (see section 2.1 of [11]). Thus, a program compiled with gcj may behave differently when run on different platforms.

4 Towards verification of floating point arithmetic

There are several problems arising when we look at floating point arithmetic with the goal of applying formal methods (namely theorem proving) to programs that use floating point numbers. The naive approach of accounting floating point arithmetic as a subset of real arithmetic will fail very quickly as we will see in the next sections. Special care has to be taken of NaNs and infinities and thus also of nearly all relations and operations.

4.1 Verification Goal

When we speak about formal verification of software, we always want to (mathematically) prove that a given piece of a program satisfies its (formal) specification. So when arguing about computations involving floating point arithmetic it is important to know, considering the design decisions of a theorem prover/library, what might be specified. To illustrate the different requirements consider the computation of the area of a triangle as shown in example 1.

Assuming the passed parameters have valid values (i.e. they are indeed the three sides of a triangle), there are several things that can be proven about this program fragment.

1. Given an upper bound for the three parameters, it is important to know if the computation overflows or involves a NaN value. Thus, a specification case should be formulated which states that, given the above requirements, neither an intermediate nor the final result is positive or negative infinity or NaN.
2. Assuming the computed result is finite, one certainly wants to prove that it represents an approximation to the mathematically exact result. Therefore, one could prove that the relative error $err = \left|\frac{\text{computed result}}{\text{exact result}}\right|$ is within a desired range (e.g. $1 \pm 2\%$). For the given example this error gets indeed very large for needle-like triangles.

Example 1

Given the three sides a, b and c of a triangle, the enclosing area A can be computed by Heron's formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

where s is half of the perimeter ($s = \frac{a+b+c}{2}$).
A direct implementation in JAVA *is given by*

— JAVA —

```
public static double triangularArea(double a,
                                    double b,
                                    double c) {
    double s = (a + b + c) / 2.;
    return Math.sqrt(s * (s-a) * (s-b) * (s-c));
}
```

— JAVA —

3. We could also try to prove whether the programs design/algorithm is correct, that is to prove that it would compute the exact area of the given triangle if all intermediate computations would have been exact. In case of this toy example this is of course not necessary or can be done by a trusted[1] computer algebra system (CAS), but considering more complex algorithms that rely on if-then-else-statements, loops or non-mathematical statements, this gets hardly done by a CAS or by hand.

From this small example it can roughly be determined what capabilities a theorem prover should provide (among many others) to handle JAVA's floating point arithmetic. These are:

1. Detecting the creation of special values (NaNs, positive and negative infinity)
2. Support for exact (real) arithmetic operations as well as for JAVA's floating point operations.
3. Support for treating JAVA's floating point arithmetic as exact (real) arithmetic.

Boldo et. al [7] propose an approach to this (for the programming language C) implemented within the Caduceus tool [9]. The authors idea is to associate three possibly different numbers with each floating-point value (variable). One for the value that is

[1] See §6 of [18] for a discussion about trustworthiness of computer algebra systems and theorem provers.

computed as by the program, another for the real value that would result if all computations were done exact and a third number representing the *ideally* computed value of the variable which means the value of the number modelled by the variable (that – in general – can be indeed different than the exactly computed number because of algorithmic approximations).

In this thesis another approach is followed. Different semantics will be defined for the symbols representing floating-point operations. Therefore, it will be possible to proof algorithmic correctness of a program without the overhead of handling rounding issues.

4.2 Semantics of Java's +, -, * and /

As already mentioned in section 3.2.4, JAVA's additive (`+` and `-`) and multiplicative (`*` and `/`) standard operators' semantics conform to the IEEE 754 standard. Thus, evaluation of addition and subtraction follows these rules (see also §15.18.2 of [15]), where the expression `a-b` is treated as `a+(-b)`:

- If `a` or `b` is NaN, the result is NaN.
- `(+`∞`) + (-`∞`)` and `(-`∞`) + (+`∞`)` is NaN.
- If `a =` $\pm\infty$ and `b` is neither NaN nor $\mp\infty$ or `b =` $\pm\infty$ and `a` is neither NaN nor $\mp\infty$, the result is $\pm\infty$.
- `(+0.0) + (-0.0)`, `(-0.0) + (+0.0)` and `(+0.0) + (+0.0)` results in `+0.0`.
- `(-0.0) + (-0.0)` results in `-0.0`.
- If `a` is nonzero, not NaN and finite and `b = -a`, the result is `+0.0`.
- In all other cases the sum is computed exactly and rounded according to IEEE 754 round-to-nearest (see section 3.2.2). If the operation is not in scope of a `strictfp` modifier (see section 3.2.5), the appropriate extended-exponent value set (see section 3.2.1.3) may be chosen as the destination format for rounding.

Similar rules apply for the multiplication `a*b` (see §15.17.1 of [15]):

- If `a` or `b` is NaN, the result is NaN.
- Multiplying a zero by an infinity results in NaN regardless of the operands' signs.
- `(+`∞`) * (`$\pm\infty$`)` and `(-`∞`) * (`$\mp\infty$`)` results in $\pm\infty$.
- If `a` is `+`∞ and `b` is finite and has a positive sign or if `a` is `-`∞ and `b` is finite and has a negative sign, the result is `+`∞.

- If `a` is +∞ and `b` is finite and has a negative sign or if `a` is -∞ and `b` is finite and has a positive sign, the result is -∞.
- The two rules above also apply for interchanging `a` with `b`.
- In all other cases the product is computed exactly and rounded according to IEEE 754 round-to-nearest. If the operation is not in scope of a `strictfp` modifier, the appropriate extended-exponent value set may be chosen as the destination format for rounding.

For the division `a / b` the rules are (see §15.17.2 of [15]):

- If `a` or `b` is NaN, the result is NaN.
- Dividing two infinities or two zeros results in NaN regardless of the operands' signs.
- If `a = +0.0` and `b` is neither NaN nor zero, the result is a zero with the sign of `b`.
- If `a = -0.0` and `b` is neither NaN nor zero, the result is a zero with the opposite sign of `b`.
- If `a = +∞` and `b` is finite, the result is an infinity with the sign of `b`.
- If `a = -∞` and `b` is finite, the result is an infinity with the opposite sign of `b`.
- if `a` is finite and nonzero and `b = +0.0`, the result is an infinity with the sign of `a`.
- if `a` is finite and nonzero and `b = -0.0`, the result is an infinity with the opposite sign of `a`.
- In all other cases the division is computed exactly and rounded according to IEEE 754 round-to-nearest. If the operation is not in scope of a `strictfp` modifier, the appropriate extended-exponent value set may be chosen as the destination format for rounding.

Before any of the operations is executed in Java, binary numeric promotion as described in §5.6.2 of [15] is performed on the operands.

4.3 Signed Zeros

As mentioned in section 3.2.1, every floating point format has two representations for the real value 0, namely `+0.0` and `-0.0`. As they both represent the same value, they compare equal (according to the IEEE 754 standard [20]). However, many floating point operations involving zeros distinguish between the two representations. So for example, the following statements are all true.

- `+0.0 == -0.0`
- `1.0 / +0.0 != 1.0 / -0.0`
  And in general: $\neg\forall f \quad f(+0.0) = f(-0.0)$
  But also of course: $\exists f \quad f(+0.0) = f(-0.0)$
- The following definition of the absolute value of `x` is undesired.
  `|x| := (x >= 0.0) ? x : -x`
  As for `x = -0.0` the comparison `x >= 0.0` evaluates to `true` and thus `|-0.0|` would be `-0.0` rather than `+0.0`.
- A correct definition of the absolute value of `x` is
  `|x| := (x <= 0.0) ? +0.0 - x : x`
  Note that `+0.0 - x` is somewhat different from `-x`, as the result for `x = +0.0` differs in its sign (see section 4.2).

4.4 NaNs

By the introduction of NaN values, equality of floating point numbers is not reflexive anymore. This is because the IEEE 754 standard declares NaNs to be *unordered* (§5.7 of [20]). That means `x == NaN` always evaluates to `false` even if `x` is NaN. The same applies for inequality (`x != NaN` always evaluates to `true` even if `x` is NaN) as well as for `>`, `<`, `<=` and `>=` (A `NaN` involved here leads to `false`).

As a result of that, there are several well known mathematical relations which won't hold if applied to floating point arithmetic. These include the following:

- $\forall$ `x,y :  x <= y` $\Leftrightarrow$ $\neg$`(x > y)`
- $\forall$ `x,y :  x >= y` $\Leftrightarrow$ $\neg$`(x < y)`

On the other hand the following relation still holds:

- $\forall$ `x,y :  x == y` $\Leftrightarrow$ $\neg$`(x != y)`

Another interesting example which combines the usage of signed zeros (see section 4.3) and NaNs is given in §4.3 of [26]. The four definitions of computing the minimum of two floating point numbers given below are all equivalent when considering real arithmetic. However, in floating point arithmetic the results differ if signed zeros or NaNs are involved.

1. `min(x,y) = x < y ? x : y`
2. `min(x,y) = x <= y ? x : y`
3. `min(x,y) = x > y ? y : x`

4. `min(x,y) = x >= y ?  y : x`

This is a serious issue concerning formal verification, as the mathematical relations in a logic should always have the mathematical semantics. Equality for example should always be reflexive, symmetric, transitive and substitutive. Thus, dealing with NaNs and signed zeros in a logic is not quite straight forward and results in additional predicates (see sections 6.1.2 and 6.2).

4.5 Rounding issues

As seen in the last two sections, many mathematical relations do not hold in floating point arithmetic. Another reason for this is that most operations suffer from rounding and are thus not exact. Therefore, as computations normally include more than one operation, errors from rounding are propagated through the whole computation and can significantly affect accuracy.

Important properties getting lost due to rounding are amongst others the following (examples are considered to be evaluated in `double` precision and decimals are rounded correctly to `double` using round-to-nearest):

- Associativity of addition and multiplication
  Examples:
  `1 == (1e50 + (-1e50)) + 1` $\neq$ `1e50 + ((-1e50) + 1) == 0`
  `0.1 * (0.2 * 0.3)` $\neq$ `(0.1 * 0.2) * 0.3`
- The distributive law
  Example:
  `(0.1 + 0.7) * 0.2` $\neq$ `0.1 * 0.2 + 0.7 * 0.2`
- Many mathematical transformations such as $\frac{x \cdot x}{x} = x \quad (x \neq 0)$
  Example: `(0.1 * 0.1) / 0.1 != 0.1`

On the other hand, due to gradual underflow (see section 3.2.1), the following is true for all floating point numbers `x`, `y` except for infinities, although the computation of `x-y` may suffer from rounding.

- `x == y` $\Leftrightarrow$ `x - y == 0`

4.6 Platform dependency issues

As JAVA is not deterministic in its floating-point arithmetic due to the fact that it allows operations being computed in the extended-exponent value sets (see section 3.2.1.3), results may vary depending on which virtual machine (VM) is used and on which platform a program runs. Furthermore, modern VMs are capable of just-in-time (JIT) compilation for frequently used code. Thus, as explained in [26], the same code lines can produce different results when run with the same parameters on the same platform using the same VM but run unequally often. This is of course toxic for thepurpose of verification. Fortunately, there is the `strictfp` modifier (see section 3.2.5) that prevents the VM from using the extended-exponent value sets for computations within the scope of the modifier. Hence, computations within the scope of `strictfp` should be deterministic (in fact there are JAVA compilers, namely the GNU JAVA Compiler, that ignores this modifier and thus does not fulfill the JAVA language specification [15] in this point, see §2.1 of [11]).

To not have to deal with indeterminism, we will consider only FP-strict expressions (those in the scope of a `strictfp` modifier) in the following.

4.7 Conclusion

The issues mentioned in sections 4.3, 4.4 and 4.5 arise from the difference of real operators' semantics and those of floating-point operations. Thus, the effort it takes to cope with them is to precisely define the floating-point operators' semantics in a theorem prover's logic according to §15.18 in [15] or section 4.2. Most of the rules given in 4.2 can be expressed as `if-then-else` statements, which take little effort to deal with in most logics. Problematic is only the last rule for each operation, involving rounding to the correct (representable) number of the desired format. This rule however is the most important one, as it is the "normal case". Thus, the problem is finding a formal description of the rounding function, which will be addressed in the next chapter.

5 Rounding

5.1 Related Work

The main task in formalizing floating-point arithmetic is to find an appropriate description of the rounding functions involved. There are several attempts to this issue. A formalization of floating-point numbers using the language Z [31] was proposed by Barrett [4] already in 1989. A second attempt was made by Miner [25] using PVS [27]. The improvement to [4] was that there is a proving environment for PVS. Harrison [17] and Russinoff [28] developed libraries for HOL [13] and ACL2 [22] with the main purpose of verifying the hardware design of the floating-point units developed at Intel™[3] and AMD™[1]. The most recent attempt was made by Daumas et al [8] who developed a generic, highly parameterized library for floating-point numbers within the Coq system [19].

The different approaches can be grouped into two categories. Miner [25] and Russinoff [28] try to "compute" the rounded result of a real number directly, whereas the others establish some criteria (in the form of relations) the rounded number must fulfill.

Section 5.2 will give an overview how the rounded result can be computed directly, section 5.3 will show how the other approach works. The last section in this chapter (section 5.4) presents the approach used in the scope of this thesis.

5.2 Modelling rounding by direct computation

Miner [25] and Russinoff [28] use this method to model the rounding function. They both make use of the floor and ceil functions from the real numbers to the integers.

First, the unrounded number is scaled according to its floating-point type precision and then rounded to an integer by either the floor or the ceil function according to the desired rounding-mode. A second step scales back the number to its original power. This approach shifts the problem of rounding to the problem of finding appropriate definitions of the floor and ceil function.

5.3 Modelling rounding by relations

Barrett [4], Harrison [17] and Daumas et al [8] use a different approach. They define the rounding function by establishing predicates the rounding function must fulfill.

Daumas et al [8] for example, define a rounding to be a relation between the real numbers and the floating-point format, which is total, compatible, monotone and a projection. According to the authors, these four properties are the key properties of a rounding function. Rounding to the nearest is then defined by a fifth predicate stating, that the rounded result must indeed have the least distance from the argument.

5.4 Rounding up and down

5.4.1 Unbounded rounding

The approach followed here is based on the work presented in [23] and [24]. The fact, that rounding down (and up) can be described uniquely by a few fundamental properties is used to build a formal description of a rounding function that has the semantics demanded by the rounding mode `round-to-nearest` (⇒ section 3.2.2).

At first, the functions $\triangledown$ and $\triangle$ are defined to be generalized forms of the rounding modes `round-to-negative-infinity` and `round-to-positive-infinity` respectively.

Definition 5.1 (Downward Rounding)
Let $\triangledown_R : \mathbb{R} \to R$ *with* $R \subset \widehat{\mathbb{F}}$ *be a mapping with the following properties*

$$\triangledown_R(a) = a \qquad \forall a \in R \tag{5.1}$$

$$a \le b \Rightarrow \triangledown_R(a) \le \triangledown_R(b) \qquad \forall a, b \in \mathbb{R} \tag{5.2}$$

$$\triangledown_R(a) \le a \qquad \forall a \in \mathbb{R} \tag{5.3}$$

then $\triangledown_R$ *is a* ***downward rounding*** *into* R*.*

Definition 5.2 (Upward Rounding)
Let $\triangle_R : \mathbb{R} \to R$ *with* $R \subset \widehat{\mathbb{F}}$ *be a mapping with the following properties*

$$\triangle_R(a) = a \qquad \forall a \in R \tag{5.4}$$

$$a \le b \Rightarrow \triangle_R(a) \le \triangle_R(b) \qquad \forall a, b \in \mathbb{R} \tag{5.5}$$

$$a \le \triangle_R(a) \qquad \forall a \in \mathbb{R} \tag{5.6}$$

then $\triangle_R$ *is an* ***upward rounding*** *into* R*.*

Note, that the range of $\bigtriangledown$ and $\bigtriangleup$ has to be a subset of $\widehat{\mathbb{F}}$. Thus, every number rounded by one of these functions is always finite (and never an infinity). Hence, the range R of $\bigtriangledown$ may not be bounded below and analogously the range of $\bigtriangleup$ may not be bounded above. Therefore, defining both roundings on the same set requires the set to be not bounded at all.

Note also, that the functions $\bigtriangledown$ and $\bigtriangleup$ are well defined by the given properties.

Next, generalized sets for the floating-point types `float` and `double` are defined.

Definition 5.3
Let S be the set containing all floating-point numbers representable with a 24-bit mantissa and let D be the set containing all floating-point numbers representable with a 53-bit mantissa.

$$\begin{aligned} S &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid -2^{24} < m < 2^{24} \land e \in \mathbb{Z}\right\} \\ D &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid -2^{53} < m < 2^{53} \land e \in \mathbb{Z}\right\} \end{aligned}$$

Now the function $\bigtriangledown_S$ ($\bigtriangledown_D$) defines a rounding according to the rounding mode `round-to-negative-infinity` but with an unbounded exponent range. Thus, every real number is rounded downwards to 24 (53) significant binary digits by the function. The same applies to $\bigtriangleup_S$ and $\bigtriangleup_D$ modelling an unbounded form of the rounding mode `round-to-positive-infinity`.

The advantage of having a simple description of downward and upward rounding is, that rounding to the nearest can be defined rather simply by using the other roundings. To properly handle the case where `round-to-nearest` results in a tie and thus the least significant bit of the rounded number has to be zero, the sets S^0 and D^0 are introduced.

Definition 5.4
Let S^0 be the set containing all floating-point numbers representable with a 23-bit mantissa and let D^0 be the set containing all floating-point numbers representable with a 52-bit mantissa.

$$\begin{aligned} S^0 &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid -2^{23} < m < 2^{23} \land e \in \mathbb{Z}\right\} \\ D^0 &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid -2^{52} < m < 2^{52} \land e \in \mathbb{Z}\right\} \end{aligned}$$

The following definition introduces a rounding function according to the rounding mode `round-to-nearest` but with unbounded exponent range.

Definition 5.5
Let R be one of $\{S, D\}$. Then $\Box_R : \mathbb{R} \to R$ is defined by

$$\Box_R(a) = \begin{cases} a & , \textit{if } a \in R \\ \nabla_R(a) & , \textit{else if } 2a < \nabla_R(a) + \triangle_R(a) \\ \triangle_R(a) & , \textit{else if } 2a > \nabla_R(a) + \triangle_R(a) \\ \nabla_R(a) & , \textit{else if } 2a = \nabla_R(a) + \triangle_R(a) \wedge \nabla_R(a) \in R^0 \\ \triangle_R(a) & , \textit{else if } 2a = \nabla_R(a) + \triangle_R(a) \wedge \triangle_R(a) \in R^0 \end{cases}$$

Remark: *As $\frac{\nabla_R(a)+\triangle_R(a)}{2}$ is exactly the middle of two consecutive floating-point numbers $f_1 = \nabla_R(a)$ and $f_2 = \triangle_R(a)$ in R such that $a \in [f_1, f_2]$, one has to check only whether a is greater or smaller than the middle to decide if f_1 or f_2 is the correctly rounded version of a.*

5.4.2 Bounded rounding

Now the definition of $\Box_R$ will be refined to meet all the requirements of the IEEE 754 `round-to-nearest` rounding mode. Therefore, the set of normalized floating-point numbers with the exponent unbounded above and the set of denormalized floating-point numbers are defined for both floating-point types.

Definition 5.6
Let S_N be the set of normalized floating-point numbers with the exponent unbounded above of type `float` and let D_N be the set of normalized floating-point numbers with the exponent unbounded above of type `double`.

$$\begin{aligned} S_N &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 2^{23} \leq |m| < 2^{24} \wedge -149 \leq e\right\} \\ D_N &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 2^{52} \leq |m| < 2^{53} \wedge -1074 \leq e\right\} \end{aligned}$$

Let S_D be the set of denormalized floating-point numbers of type `float` and let D_D be the set of denormalized floating-point numbers of type `double`.

$$\begin{aligned} S_D &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \leq |m| < 2^{23} \wedge e = -149\right\} \\ D_D &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \leq |m| < 2^{52} \wedge e = -1074\right\} \end{aligned}$$

*The set $R_C = R_N \cup R_D$ for $R \in \{S, D\}$ is called the set of **canonical** floating-point numbers of the appropriate type.*

Remark: *As R_C is unbounded, the downward rounding $\bigtriangledown_{R_C}$ and the upward rounding $\bigtriangleup_{R_C}$ are defined by definitions 5.1 and 5.2.*

Next, the set of all canonical floating-point numbers with the least significant bit zero has to be defined.

Definition 5.7
The sets S_C^0 and D_C^0 are defined by

$$
\begin{aligned}
S_C^0 &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 2^{23} \le |m| < 2^{24} \wedge -149 \le e \wedge \exists k \in \mathbb{Z} : \frac{m}{2} = k\right\} \\
&\quad \cup \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \le |m| < 2^{23} \wedge e = -149 \wedge \exists k \in \mathbb{Z} : \frac{m}{2} = k\right\} \\
&= \left\{\widehat{(k,e)} \in \widehat{\mathbb{F}} \mid 2^{22} \le |k| < 2^{23} \wedge -148 \le e\right\} \\
&\quad \cup \left\{\widehat{(k,e)} \in \widehat{\mathbb{F}} \mid 0 \le |k| < 2^{22} \wedge e = -148\right\} \\
&= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \le |m| < 2^{23} \wedge -148 \le e\right\} \\
D_C^0 &:= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 2^{52} \le |m| < 2^{53} \wedge -1074 \le e \wedge \exists k \in \mathbb{Z} : \frac{m}{2} = k\right\} \\
&\quad \cup \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \le |m| < 2^{52} \wedge e = -1074 \wedge \exists k \in \mathbb{Z} : \frac{m}{2} = k\right\} \\
&= \left\{\widehat{(k,e)} \in \widehat{\mathbb{F}} \mid 2^{51} \le |k| < 2^{52} \wedge -1073 \le e\right\} \\
&\quad \cup \left\{\widehat{(k,e)} \in \widehat{\mathbb{F}} \mid 0 \le |k| < 2^{51} \wedge e = -1073\right\} \\
&= \left\{\widehat{(m,e)} \in \widehat{\mathbb{F}} \mid 0 \le |m| < 2^{52} \wedge -1073 \le e\right\}
\end{aligned}
$$

These sets have the property that for any $a \in R_C$ either $\bigtriangledown_{R_C}(a)$ or $\bigtriangleup_{R_C}(a)$ is in R_C^0. Thus, definition 5.5 can also be applied to R_C and the resulting function $\Box_{R_C}$ defines the correct rounding for all numbers which would not raise an IEEE 754 overflow exception (see section 3.2.3). So, it is now possible to define a rounding function according to the rounding mode `round-to-nearest`.

Definition 5.8
Let SV be the set of IEEE 754 special values, $SV := \{NaN, +\infty, -\infty, +0.0, -0.0\}$. The function ***round****$_S$ $: \mathbb{R} \cup SV \to S_C \cup SV$ *modelling the mode* ***round-to-nearest*** *for*

the floating-point type `float` *is defined by*

$$\texttt{round}_S(a) := \begin{cases} a & \text{, if } a \in \{NaN, +\infty, -\infty, +0.0, -0.0\} \\ +\infty & \text{, else if } a \geq 2^{127}(2 - 2^{-24}) \\ -\infty & \text{, else if } a \leq -2^{127}(2 - 2^{-24}) \\ +0.0 & \text{, else if } a \geq 0 \wedge a \leq 2^{-150} \\ -0.0 & \text{, else if } a < 0 \wedge a \geq -2^{-150} \\ \Box_{S_C}(a) & \text{, otherwise.} \end{cases}$$

Analogously the function $\texttt{round}_D : \mathbb{R} \cup SV \to S_C \cup SV$ *for modelling the rounding for the type* `double` *is defined by*

$$\texttt{round}_D(a) := \begin{cases} a & \text{, if } a \in \{NaN, +\infty, -\infty, +0.0, -0.0\} \\ +\infty & \text{, else if } a \geq 2^{1023}(2 - 2^{-53}) \\ -\infty & \text{, else if } a \leq -2^{1023}(2 - 2^{-53}) \\ +0.0 & \text{, else if } a \geq 0 \wedge a \leq 2^{-1075} \\ -0.0 & \text{, else if } a < 0 \wedge a \geq -2^{-1075} \\ \Box_{D_C}(a) & \text{, otherwise.} \end{cases}$$

Using these functions in a deductive system like the KeY system [30], the underlying logic should be able to handle the case distinctions pretty well. The interesting case, where the rounded result is not a special value, is reduced to either a downward rounding or an upward rounding which can be handled by the three properties from definitions 5.1 and 5.2. These properties are also rather simple to formulate in a first order logic.

6 The logic part

In this chapter the logic and calculus introduced by [6] will be extended to cover all essential changes needed to reduce the verification effort of problems concerning Java floating-point arithmetic to those concerning only real arithmetic. That means, proof obligations resulting from a specification of a Java program containing floating-point operations will be reduced to proof obligations of first order logic with real arithmetic.

The logic JavaCardDL introduced by [6] is a dynamic logic where JavaCard program fragments may appear inside modalities. The calculus for JavaCardDL is a sequent calculus that mimics symbolic execution of the JavaCard fragments inside the modalities.

As JavaCardDL concerns only about JavaCard programs, but floating-point arithmetic is not part of JavaCard, the logic introduced here will cover a subset of Java, that contains all JavaCard features and additionally all floating-point features from Java. The resulting logic will be called JavaCardDL$^+$.

6.1 Extensions to JavaCardDL

In the following, symbols and denotations from [6] chapter 3 are used.

6.1.1 Type hierarchy

A JavaCardDL$^+$ type hierarchy is defined on top of a JavaCardDL type hierarchy from [6] (definition 3.2). It is important to note that the definition differentiates between *dynamic* and *abstract* types. This distinction becomes relevant when defining the semantics of the logic (see section 6.1.4).

Definition 6.1 (JavaCardDL$^+$ type hierarchy)
A JavaCardDL$^+$ *type hierarchy is a* JavaCardDL *type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *(⇒ definition 3.2 [6]), where*

- $\mathcal{T}_d$ *contains (additionally to definition 3.2 [6]) at least the types* `binFloat`*,* `real` *and* `SVDomain`*,*
- $\mathcal{T}_a$ *contains at least the types* `jfloat`*,* `jdouble`*,* `javaFloat` *and* `numeric`*,*
- `integer` $\sqsubseteq^0$ `binFloat`*,*
- `binFloat` $\sqsubseteq^0$ `real`*,*
- `binFloat` $\sqsubseteq^0$ `SVDomain`*,*
- `SVDomain` $\sqsubseteq^0$ A *and* $A \sqsubseteq^0$ `javaFloat` *for* $A \in \{$`jfloat`, `jdouble`$\}$*,*
- `javaFloat` $\sqsubseteq^0$ `numeric`*,*
- `real` $\sqsubseteq^0$ `numeric` *and*
- $A \sqcap B = \bot$ *for all* $A \in \{$`binFloat`, `real`, `SVDomain`, `jfloat`, `jdouble`, `javaFloat`$\}$ *and* $B \sqsubseteq$ `Object`*.*

The part of a JavaCardDL$^+$ type hierarchy relevant for reasoning about floating-point arithmetic is depicted in figure 6.1. An arrow from A to B indicates that A is a subtype of B. The dashed arrow from $\bot$ to `integer` indicates that intermediate types are omitted here.

6.1.2 Signature

The definition of the signature is also given in [6] (definition 3.4). But, the sets of *rigid* function and predicate symbols must contain additional elements. A *rigid* symbol has the same value in all states of the (dynamic) logic, whereas *non-rigid* symbols may "change" their value from state to state (see section 6.1.4) .

Definition 6.2 (JavaCardDL$^+$ signature)
A JavaCardDL$^+$ *signature for a given* JavaCardDL$^+$ *type hierarchy is a* JavaCardDL *signature (definition 3.4 [6]), where the set* $FSym_r^0$ *of rigid functions also contains the symbols given in tables 6.1, 6.2, 6.3 and 6.4. And, the set* $PSym_r^0$ *of rigid predicates contains also the symbols given in table 6.5. The typing for those symbols is also given in the tables.*

Informally, the symbols of table 6.1 reflect format specific attributes of the floating-point types. There are constants for the special values NaN, signed zeros and infinities as well as for every real number. The function R maps two integers m and e to the floating-point number represented by $\widehat{(m,e)} \in \widehat{\mathbb{F}}$ (see section 2.2), whereas the functions `precision`, `maxMan`, `maxExp` and `minExp` define the exact format of a floating-point type.

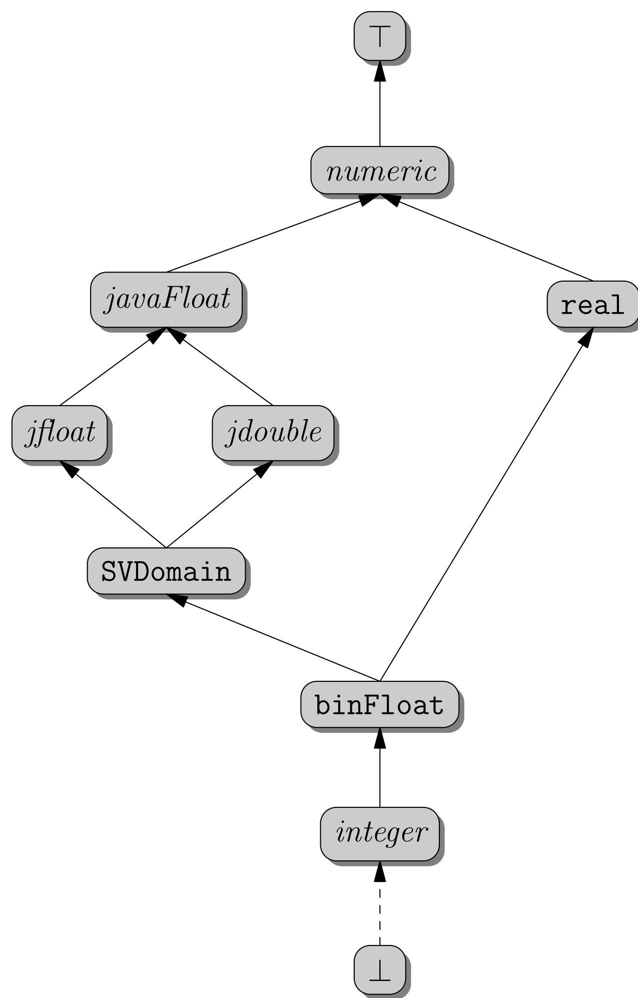

Figure 6.1. Basic JavaCardDL$^+$ type hierarchy

function	typing
NaN	javaFloat
$+\infty$	javaFloat
$-\infty$	javaFloat
+0.0	javaFloat
-0.0	javaFloat
R	integer $\times$ integer $\rightarrow$ numeric
for each $T \in \{\texttt{jfloat}, \texttt{jdouble}\}$:	
$\texttt{precision}_T$	integer
$\texttt{maxMan}_T$	integer
$\texttt{maxExp}_T$	integer
$\texttt{minExp}_T$	integer

Table 6.1. Format specific rigid functions

function	typing
$+_r$	real $\times$ real $\rightarrow$ real
$-_r$	real $\times$ real $\rightarrow$ real
$*_r$	real $\times$ real $\rightarrow$ real
$/_r$	real $\times$ real $\rightarrow$ real
$\%_r$	real $\times$ real $\rightarrow$ real
$\texttt{neg}_r$	real $\rightarrow$ real
for each $r \in \mathbb{R}$: r	real

Table 6.2. Functions for real arithmetic

function	typing
$+_j$	numeric $\times$ numeric $\rightarrow$ numeric
$-_j$	numeric $\times$ numeric $\rightarrow$ numeric
$*_j$	numeric $\times$ numeric $\rightarrow$ numeric
$/_j$	numeric $\times$ numeric $\rightarrow$ numeric
$\%_j$	numeric $\times$ numeric $\rightarrow$ numeric
$\texttt{neg}_j$	numeric $\rightarrow$ numeric

Table 6.3. Functions for exact floating-point arithmetic

function	typing
	for each $T \in \{\texttt{jfloat}, \texttt{jdouble}\}$:
$\texttt{addJ}_T$	$\texttt{numeric} \times \texttt{numeric} \to \texttt{numeric}$
$\texttt{subJ}_T$	$\texttt{numeric} \times \texttt{numeric} \to \texttt{numeric}$
$\texttt{mulJ}_T$	$\texttt{numeric} \times \texttt{numeric} \to \texttt{numeric}$
$\texttt{divJ}_T$	$\texttt{numeric} \times \texttt{numeric} \to \texttt{numeric}$
$\texttt{modJ}_T$	$\texttt{numeric} \times \texttt{numeric} \to \texttt{numeric}$
$\texttt{negJ}_T$	$\texttt{numeric} \to \texttt{numeric}$
$\texttt{castToJ}_T$	$\texttt{numeric} \to \texttt{numeric}$
$\texttt{round}_T$	$\texttt{numeric} \to \texttt{javaFloat}$
$\Box_T$	$\texttt{numeric} \to \texttt{javaFloat}$
$\triangle_T$	$\texttt{numeric} \to \texttt{javaFloat}$
∇_T	$\texttt{numeric} \to \texttt{javaFloat}$

Table 6.4. Functions modelling JAVA operations

predicate	typing
$<_r$	$\texttt{real} \times \texttt{real}$
$\leq_r$	$\texttt{real} \times \texttt{real}$
$>_r$	$\texttt{real} \times \texttt{real}$
$\geq_r$	$\texttt{real} \times \texttt{real}$
$=_j$	$\texttt{numeric} \times \texttt{numeric}$
$\neq_j$	$\texttt{numeric} \times \texttt{numeric}$
$<_j$	$\texttt{numeric} \times \texttt{numeric}$
$\leq_j$	$\texttt{numeric} \times \texttt{numeric}$
$>_j$	$\texttt{numeric} \times \texttt{numeric}$
$\geq_j$	$\texttt{numeric} \times \texttt{numeric}$
inFloat	numeric
inDouble	numeric

Table 6.5. Predicates for real and floating-point arithmetic

The functions given in table 6.2 reflect the common arithmetic operators on $\mathbb{R}$. In table 6.3 however, the functions with subscript j have semantics reflecting an exact arithmetic on floating-point numbers.

Table 6.4 summarizes function symbols reflecting the operators of JAVA. The intended semantics of these will change according to the verification task. If only algorithmic correctness is of interest (compare to section 4.1), one certainly wants to ignore rounding issues as they are very "expensive", therefore these symbols will have different semantics in different calculi. When exact JAVA semantics are desired, the functions ending with $\mathtt{J}_T$ will have the semantics of the appropriate JAVA operations and $\mathtt{round}_T$ will model the IEEE 754 rounding mode *round-to-nearest* (see section 3.2.2).

The predicates introduced in the upper part of table 6.5 have the common mathematical semantics on the real numbers. The predicates with subscript j however, will additionally deal with the special values reflecting JAVA's semantic of the corresponding operators. Note that the formula $\mathtt{NaN} \doteq \mathtt{NaN}$ will be *true* in JAVACARDDL$^+$, the formula $\mathtt{NaN} =_j \mathtt{NaN}$ however will be *false*. The predicates `inFloat` and `inDouble` will have different semantics in different calculi as the functions of table 6.4. In case of JAVA semantics the predicates are only valid if (and only if) the given argument would be representable in the appropriate JAVA type.

6.1.3 Syntax

The definition of the syntax of JAVACARDDL$^+$ does not differ from the syntax definition for JAVACARDDL. The definitions for the syntax of terms (definition 3.7, [6]), syntactic updates (definition 3.8, [6]) and formulae (definition 3.14, [6]) are applied to a JAVACARDDL$^+$ type hierarchy and JAVACARDDL$^+$ signature to obtain the JAVACARDDL$^+$ syntax.

JAVACARDDL as well as JAVACARDDL$^+$ uses the concept of *updates* to model state transitions instead of expressing them by syntactic substitutions and equations. Informally speaking, an update describes the changes to the program variables during symbolic execution and thus models the changes of the memory heap. There are several advantages of updates compared to substitutions. One of them is that updates can be simplified before applied to a formula which results in fewer proof splits and a shorter proof in general. More about updates can be found in section 3.2.3 of the KeY-Book ([6]).

6.1.4 Semantics

As indicated above, JavaCardDL$^+$ comes in different flavors concerning the semantic. First of all, a semantic $\mathcal{S}_{\text{JAVA}}$ is defined, that reflects the Java semantics exactly. A second semantics $\mathcal{S}_{\mathtt{ALG}}$ is defined, which ignores rounding-issues and the special values NaN, positive infinity and negative infinity. It is intended to use $\mathcal{S}_{\mathtt{ALG}}$ for verifying algorithmic correctness (see also section 4.1).

Like in [6], the semantics are based on *Kripke structures* which contain a partial first-order model fixing the interpretation of rigid symbols. Since both semantics $\mathcal{S}_{\text{JAVA}}$ and $\mathcal{S}_{\mathtt{ALG}}$ only differ in the interpretation of some (rigid) symbols introduced above (see table 6.4 and 6.5), it is just needed to adapt the definition of a JavaCardDL Kripke structure to fix the interpretation of these symbols.

Definition 6.3 (JavaCardDL$^+$ Kripke structure)
Given a JavaCardDL$^+$ *type hierarchy, a* JavaCardDL$^+$ *signature and a normalized* JavaCard$^+$ *program* [1], *a* JavaCardDL$^+$ *Kripke structure is a* JavaCardDL *Kripke structure* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *where the partial domain* $\mathcal{D}_0$ *of the partial model* $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ *also satisfies*

- $\mathcal{D}_0^{\mathtt{SVDomain}} = \{NaN, +\infty, -\infty, +0.0, -0.0\}$,
- $\mathcal{D}_0^{\mathtt{real}} = \mathbb{R}$ *and*
- $\mathcal{D}_0^{\mathtt{binFloat}} = \widehat{\mathbb{F}}$.

The partial interpretation $\mathcal{I}_0$ *must satisfy the constraints given in appendix A. Note, that the different semantics are taken into account here.*

Based on JavaCardDL$^+$ Kripke structures, the semantics of terms, updates and formulae of $\mathcal{S}_{\text{JAVA}}$ and $\mathcal{S}_{\mathtt{ALG}}$ are defined by definitions 3.27, 3.31 and 3.34 of [6].

6.2 A calculus for JavaCardDL+

A calculus for JavaCardDL$^+$ is achieved by extending the calculus for JavaCardDL defined in sections 3.4 to 3.9 in [6]. Although, few rules of the JavaCardDL calculus must be revised in order to be sound within JavaCardDL$^+$ semantics. These particularly include the basic assignment and conditional rules (sections 3.6.1 and 3.6.3 in [6]).

[1] A normalized JavaCard$^+$ program is a normalized JavaCard program P (definition 3.10, [6]), where P may also contain legal Java statements involving floating-point types and operations.

The assignment rules are incorrect for floating-point expressions, as implicit conversion of an integral number to a floating-point format may result in loss-of-precision (see §5.2 and §5.1.2 of the JAVALanguage Specification [15]). Similar to that, JAVA's equality comparison operator does not coincide with the logic's equality relation (see section 4.4: `NaN == NaN` evaluates to *false*).

In the following let π be a non-active program prefix and ω a program suffix, according to [6].

6.2.1 A new assignment rule

The assignment rule given in [6] is applicable for arbitrary types (especially if the type of the left hand side is not equal to the type on the right hand side of the assignment). This is ok for programs not involving floating-point arithmetic as only a widening primitive conversion (which is allowed in an assignment, §5.2 [15]) may change the actual value of the expression on the right hand side of an assignment. And a widening primitive conversion of **integral** types is always exact.

A simple solution to get a sound assignment rule is to restrict the application of the rule to those cases where the right hand side expression has exactly the same type as the left hand side. However, a compile-time correct assignment of different types must then be handled separately. Thus, the (schematic) assignment rule is restricted to

$$\frac{\Longrightarrow \{loc_T := val_T\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ loc_T = val_T;\ \omega\rangle\phi}\ \textbf{assignment}$$

Where the schematic variables loc_T and val_T must be instantiated by simple ($\widehat{=}$ free of side-effects) expressions of the same type. A second rule is needed covering the case where an implicit conversion to the left hand side's type is needed.

$$\frac{\Longrightarrow \langle\pi\ loc_{T1} = (T1)val_{T2};\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ loc_{T1} = val_{T2};\ \omega\rangle\phi}\ \textbf{assignment_cast}$$

Simplification of the resulting expression needs further rules according to the types involved. If only reference types are involved, the resulting update will just be the same as above. Thus, the resulting rule will be

$$\frac{\Longrightarrow \{loc_{T1} := val_{T2}\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ loc_{T1} = (T1)val_{T2};\ \omega\rangle\phi}\ \textbf{assignment_reference}$$

where loc_{T1} and val_{T2} are instantiated by simple expressions of some reference types. Similar the rule for assignments involving only integral types is

$$\frac{\Longrightarrow \{loc_{T1} := val_{T2}\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ loc_{T1} = (T1)val_{T2};\ \omega\rangle\phi}\ \textbf{assignment_integral}$$

where loc_{T1} and val_{T2} are both instantiated by simple expressions of primitive integral types. If loc_{T1} is instantiated by a simple expression of type `float` or `double`, the assignment rule is

$$\frac{\Longrightarrow \{loc_{T1} := \mathtt{castToJ}_{T1}(val_{T2)})\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ loc_{T1} = (T1)val_{T2};\ \omega\rangle\phi}\ \textbf{assignment_floating}$$

where $T1$ in $\mathtt{castToJ}_{T1}$ is `jfloat` if the type of loc_{T1} is `float` and it is `jdouble` if the type of loc_{T1} is `double`.

6.2.2 Program transformation and simplification rules

This section summarizes the rules needed to transform the program within a modality to an update. The rules for simplifying compound floating-point expressions are the same as for integer arithmetic as elaborated by Schlager [29]. Thus, only the rules handling simple floating-point expressions are considered here.

In the following let *slhs* be a schematic variable which can be instantiated only by a simple expression of type `float` or `double`. The schematic variables $srhs1$ and $srhs2$ can be simple expressions of arbitrary numeric types.

$$\frac{\Longrightarrow \{slhs := \mathtt{addJ}_T(\mathtt{castToJ}_{T1}(srhs1), \mathtt{castToJ}_{T1}(srhs2))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = srhs1 + srhs2;\ \omega\rangle\phi}\ \textbf{simple_add}$$

The rule *simple_add*, where $T1$ is the promoted type (see §5.6 [15]) of the expression instantiated for $srhs1$ and $srhs2$ and T is the type of the expression instantiated for *slhs*, is handling a floating-point addition. Similar, the following rules handle subtraction, multiplication, division, remainder and unary minus.

$$\frac{\Longrightarrow \{slhs := \mathtt{subJ}_T(\mathtt{castToJ}_{T1}(srhs1), \mathtt{castToJ}_{T1}(srhs2))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = srhs1 - srhs2;\ \omega\rangle\phi}\ \textbf{simple_sub}$$

$$\frac{\Longrightarrow \{slhs := \mathtt{mulJ}_T(\mathtt{castToJ}_{T1}(srhs1), \mathtt{castToJ}_{T1}(srhs2))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = srhs1 * srhs2;\ \omega\rangle\phi}\ \textbf{simple_mul}$$

$$\frac{\Longrightarrow \{slhs := \mathtt{divJ}_T(\mathtt{castToJ}_{T1}(srhs1), \mathtt{castToJ}_{T1}(srhs2))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = srhs1/srhs2;\ \omega\rangle\phi}\ \textbf{simple_div}$$

$$\frac{\Longrightarrow \{slhs := \mathtt{modJ}_T(\mathtt{castToJ}_{T1}(srhs1), \mathtt{castToJ}_{T1}(srhs2))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = srhs1\%srhs2;\ \omega\rangle\phi}\ \textbf{simple_mod}$$

$$\frac{\Longrightarrow \{slhs := \mathtt{negJ}_T(\mathtt{castToJ}_{T1}(srhs1))\}\langle\pi\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = -srhs1;\ \omega\rangle\phi}\ \textbf{simple_neg}$$

The cast operator is handled by the assignment rule introduced above. However, if a nonsimple expression nse shall be casted, the rule needed is

$$\frac{\Longrightarrow \langle T_{nse}\ v_0 = nse;\ slhs = (T)v_0;\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ slhs = (T)nse;\ \omega\rangle\phi}\ \textbf{cast_nonsimple}$$

where T_{nse} is the type of the expression nse and v_0 is a freshly introduced variable.

Similar to the approach followed to get correct handling of assignment expressions, conditional expressions must be handled. The "old" rules of the JavaCardDL calculus are valid when no floating-point types are involved in the guard of a conditional statement. Whenever an expression on one side of a `==`, `!=`, `<`, `<=`, `>` or `>=` operator is of floating-point type, the operator must be "translated" to the according predicate with subscript j of table 6.5. An if-then-else statement for example is handled by the following rule, if at least one of $slhs$ and $srhs$ is instantiated by a floating-point expression. The type T is the promoted type of $slhs$ and $srhs$.

$$\frac{\Longrightarrow \mathtt{\backslash if}(\mathtt{castToJ}_T(slhs) =_j \mathtt{castToJ}_T(srhs))\ \mathtt{\backslash then}\langle\pi\ p_1;\ \omega\rangle\phi\ \mathtt{\backslash else}\langle\pi\ p_2;\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ \textbf{if}(slhs == srhs)\ \textbf{then}\ p_1;\ \textbf{else}\ p_2;\ \omega\rangle\phi}$$

6.2.3 Rules for Java semantics

While all the rules in the previous sections are inference-rules, the rules given in this and the following sections will be term-rewrite-rules. In this thesis, a term-rewrite-rule will be written like an inference-rule, where the lower part is the term that may be replaced by the upper part of the rule.

In this section rewrite-rules are presented that conform to the semantics $\mathcal{S}_{\text{JAVA}}$.

Let T be a schematic variable which is either `jfloat` or `jdouble`, then the basic JAVA operations are broken down to exact floating-point arithmetic by the following rules.

$$\frac{\texttt{round}_T(a +_j b)}{\texttt{addJ}_T(a, b)}\ \textbf{addition}$$

$$\frac{\texttt{round}_T(a -_j b)}{\texttt{subJ}_T(a, b)}\ \textbf{subtraction}$$

$$\frac{\texttt{round}_T(a *_j b)}{\texttt{mulJ}_T(a, b)}\ \textbf{multiplication}$$

$$\frac{\texttt{round}_T(a /_j b)}{\texttt{divJ}_T(a, b)}\ \textbf{division}$$

As modulo and unary minus are always exact, they do not need to be rounded. Therefore, the rules for those are

$$\frac{a \,\%_j\, b}{\texttt{modJ}_T(a, b)}\ \textbf{remainder}$$

$$\frac{\texttt{neg}_j(a)}{\texttt{negJ}_T(a)}\ \textbf{negation}$$

A cast to a floating-point type is just an appropriate rounding

$$\frac{\texttt{round}_T(a)}{\texttt{castToJ}_T(a)}\ \textbf{cast}$$

According to section 5.4, the function `round` is specified by upwards and downwards rounding $\triangle$ and $\triangledown$. Taking over- and underflow into account, $\texttt{round}_{\texttt{jfloat}}$ and $\texttt{round}_{\texttt{jdouble}}$ can be rewritten by the following rules.

$$\frac{\begin{array}{l}\texttt{\textbackslash if}\ (\texttt{inFloat}(a))\\ \texttt{\textbackslash then}\quad (a)\\ \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \geq_j 2^{127}(2-2^{-24}))\\ \qquad\qquad \texttt{\textbackslash then}\quad (+\infty)\\ \qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \leq_j -2^{127}(2-2^{-24}))\\ \qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (-\infty)\\ \qquad\qquad\qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \geq_j 0 \land a \leq_j 2^{-150})\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (+0.0)\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a <_j 0 \land a \geq_j -2^{-150})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (-0.0)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash else}\quad (\Box_{\texttt{jfloat}}(a))\\ ))))\end{array}}{\texttt{round}_{\texttt{jfloat}}(a)}\ \textbf{round_float}$$

$$\frac{\begin{array}{l}\texttt{\textbackslash if}\ (\texttt{inDouble}(a))\\ \texttt{\textbackslash then}\quad (a)\\ \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \geq_j 2^{1023}(2-2^{-53}))\\ \qquad\qquad \texttt{\textbackslash then}\quad (+\infty)\\ \qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \leq_j -2^{1023}(2-2^{-53}))\\ \qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (-\infty)\\ \qquad\qquad\qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a \geq_j 0 \land a \leq_j 2^{-1075})\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (+0.0)\\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash else}(\quad \texttt{\textbackslash if}(a <_j 0 \land a \geq_j -2^{-1075})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash then}\quad (-0.0)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\textbackslash else}\quad (\Box_{\texttt{jdouble}}(a))\\ ))))\end{array}}{\texttt{round}_{\texttt{jdouble}}(a)}\ \textbf{round_double}$$

According to definitions 5.5 to 5.7, the function $\Box_T$ can be rewritten by the rules

$$\frac{\begin{array}{l}\texttt{\textbackslash if}\ (\qquad 2a <_j \triangle_{\texttt{jfloat}}(a) + \triangledown_{\texttt{jfloat}}(a)\\ \qquad\quad \lor(\quad 2a =_j \triangle_{\texttt{jfloat}}(a) + \triangledown_{\texttt{jfloat}}(a)\\ \qquad\qquad\quad \land(\exists m, e \in \mathbb{Z} |\quad -2^{23} < m < 2^{23} \land e \geq -148\\ \qquad\qquad\qquad\qquad\qquad\qquad \land \widehat{(m,e)} = \triangledown_{\texttt{jfloat}}(a))))\\ \texttt{\textbackslash then}(\triangledown_{\texttt{jfloat}}(a))\\ \texttt{\textbackslash else}(\triangle_{\texttt{jfloat}}(a))\end{array}}{\Box_{\texttt{jfloat}}(a)}\ \textbf{rnd_float}$$

$$\dfrac{\begin{array}{ll} \texttt{\textbackslash if (} & 2a <_j \triangle_{\texttt{jdouble}}(a) + \triangledown_{\texttt{jdouble}}(a) \\ \quad \vee(& 2a =_j \triangle_{\texttt{jdouble}}(a) + \triangledown_{\texttt{jdouble}}(a) \\ & \wedge(\exists m, e \in \mathbb{Z} | \quad -2^{52} < m < 2^{52} \wedge e \geq -1073 \\ & \qquad \wedge\widehat{(m,e)} = \triangledown_{\texttt{jdouble}}(a)))) \\ \texttt{\textbackslash then}(\triangledown_{\texttt{jdouble}}(a)) & \\ \texttt{\textbackslash else}(\triangle_{\texttt{jdouble}}(a)) & \end{array}}{\Box_{\texttt{jdouble}}(a)} \ \textbf{rnd_double}$$

Both functions $\triangledown_T$ and $\triangle_T$ are axiomized by the properties (5.1) to (5.6). Thus, for each type an inference-rule is given.

$$\dfrac{\begin{array}{ll} \Gamma, & \\ \forall a \in \texttt{numeric} & ((\exists m, e \in \mathbb{Z} | \quad -2^{24} < m < 2^{24} \wedge e \geq -149 \\ & \qquad \wedge\widehat{(m,e)} = a) \\ & \rightarrow (\triangledown_{\texttt{jfloat}}(a) = a \wedge \triangle_{\texttt{jfloat}}(a) = a)), \\ \forall a, b \in \texttt{numeric} & (a \leq_r b \\ & \rightarrow (\triangledown_{\texttt{jfloat}}(a) \leq_r \triangledown_{\texttt{jfloat}}(b) \wedge \triangle_{\texttt{jfloat}}(a) \leq_r \triangle_{\texttt{jfloat}}(b))), \\ \forall a \in \texttt{numeric} & (\triangledown_{\texttt{jfloat}}(a) \leq_r a \wedge a \leq_r \triangle_{\texttt{jfloat}}(a)) \\ \Longrightarrow \Delta & \end{array}}{\Gamma \Longrightarrow \Delta}$$

$$\dfrac{\begin{array}{ll} \Gamma, & \\ \forall a \in \texttt{numeric} & ((\exists m, e \in \mathbb{Z} | \quad -2^{53} < m < 2^{53} \wedge e \geq -1074 \\ & \qquad \wedge\widehat{(m,e)} = a) \\ & \rightarrow (\triangledown_{\texttt{jdouble}}(a) = a \wedge \triangle_{\texttt{jdouble}}(a) = a)), \\ \forall a, b \in \texttt{numeric} & (a \leq_r b \\ & \rightarrow (\triangledown_{\texttt{jdouble}}(a) \leq_r \triangledown_{\texttt{jdouble}}(b) \wedge \triangle_{\texttt{jdouble}}(a) \leq_r \triangle_{\texttt{jdouble}}(b))), \\ \forall a \in \texttt{numeric} & (\triangledown_{\texttt{jdouble}}(a) \leq_r a \wedge a \leq_r \triangle_{\texttt{jdouble}}(a)) \\ \Longrightarrow \Delta & \end{array}}{\Gamma \Longrightarrow \Delta}$$

6.2.4 Rules for algorithmic semantics

The rules for the semantics $\mathcal{S}_{\texttt{ALG}}$ are much simpler, as rounding is not an issue here and thus is left underspecified (as the symbols for rounding should never occur within

a proof). The JAVA operators are just mapped to the corresponding exact operations and the function $\texttt{castToJ}_T$ is the identity.

$$\frac{a +_j b}{\texttt{addJ}_T(a, b)}\ \textbf{addition}$$

$$\frac{a -_j b}{\texttt{subJ}_T(a, b)}\ \textbf{subtraction}$$

$$\frac{a *_j b}{\texttt{mulJ}_T(a, b)}\ \textbf{multiplication}$$

$$\frac{a\ /_j\ b}{\texttt{divJ}_T(a, b)}\ \textbf{division}$$

$$\frac{a\ \%_j\ b}{\texttt{modJ}_T(a, b)}\ \textbf{remainder}$$

$$\frac{\texttt{neg}_j(a)}{\texttt{negJ}_T(a)}\ \textbf{negation}$$

$$\frac{a}{\texttt{castToJ}_T(a)}\ \textbf{cast}$$

As the symbols for rounding are left underspecified there are no rewrite-rules for them in the semantics $\mathcal{S}_{\texttt{ALG}}$.

6.2.5 Rules for Handling exact floating-point operations

Assuming, that there is a calculus for real arithmetic (or – within the KeY system – an external decision procedure that can handle real arithmetic), the only thing missing are rewrite-rules for the exact floating-point operations. Those are simply rules handling the cases where the result will be a special value and using real arithmetic otherwise. Thus, the rules are just a big if-then-else cascade, where in the last case (if no special values are involved) the operation is mapped to the corresponding operation on the reals.

Thus, the rewrite-rule for the exact addition of two floating-point numbers is given by:

$$
\frac{
\begin{array}{lllllll}
\texttt{\textbackslash if} & \multicolumn{6}{l}{(a = NaN \lor b = NaN \lor (a = \texttt{neg}_j(b) \land (a = +\infty \lor a = -\infty)))} \\
\texttt{\textbackslash then} & \multicolumn{6}{l}{(NaN)} \\
\texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{5}{l}{(a = +\infty \lor b = +\infty)} \\
 & \texttt{\textbackslash then} & \multicolumn{5}{l}{(+\infty)} \\
 & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{4}{l}{(a = -\infty \lor b = -\infty)} \\
 & & \texttt{\textbackslash then} & \multicolumn{4}{l}{(-\infty)} \\
 & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{3}{l}{(a = -0.0 \land b = -0.0)} \\
 & & & \texttt{\textbackslash then} & \multicolumn{3}{l}{(-0.0)} \\
 & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{2}{l}{(a =_j (\texttt{neg}_j(b)))} \\
 & & & & \texttt{\textbackslash then} & \multicolumn{2}{l}{(+0.0)} \\
 & & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & (a =_j 0) \\
 & & & & & \texttt{\textbackslash then} & (b) \\
 & & & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if}(b =_j 0) \\
 & & & & & & \texttt{\textbackslash then}(a) \\
 & & & & & & \texttt{\textbackslash else}(a +_r b)))))))
\end{array}
}{a +_j b}\ \textbf{exact_add}
$$

The rule for negation is straight-forward:

$$
\frac{
\begin{array}{lllll}
\texttt{\textbackslash if} & \multicolumn{4}{l}{(a = NaN)} \\
\texttt{\textbackslash then} & \multicolumn{4}{l}{(NaN)} \\
\texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{3}{l}{(a = +\infty)} \\
 & \texttt{\textbackslash then} & \multicolumn{3}{l}{(-\infty)} \\
 & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{2}{l}{(a = -\infty)} \\
 & & \texttt{\textbackslash then} & \multicolumn{2}{l}{(+\infty)} \\
 & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & (a = +0.0) \\
 & & & \texttt{\textbackslash then} & (-0.0) \\
 & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if}(a = -0.0)) \\
 & & & & \texttt{\textbackslash then}(+0.0) \\
 & & & & \texttt{\textbackslash else}(\texttt{neg}_r(b))))))
\end{array}
}{\texttt{neg}_j(a)}\ \textbf{exact_neg}
$$

The rule for exact subtraction is rather simple, as it can be expressed by exact addition and negation:

$$
\frac{a +_j \texttt{neg}_j(b)}{a -_j b}\ \textbf{exact_sub}
$$

Multiplication and Division are a bit more complex (compare to section 3.2.4). The rules for those are:

exact_mul:

$$
\frac{
\begin{array}{llllll}
\texttt{\textbackslash if} & \multicolumn{5}{l}{(a = NaN \vee b = NaN} \\
 & \multicolumn{5}{l}{\vee (a = +\infty \wedge b =_j 0) \vee (a = -\infty \wedge b =_j 0)} \\
 & \multicolumn{5}{l}{\vee (b = +\infty \wedge a =_j 0) \vee (b = -\infty \wedge a =_j 0))} \\
\texttt{\textbackslash then} & \multicolumn{5}{l}{(NaN)} \\
\texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{4}{l}{((a = +\infty \wedge b >_j 0) \vee (a = -\infty \wedge b <_j 0)} \\
 & & \multicolumn{4}{l}{\vee (b = +\infty \wedge a >_j 0) \vee (b = -\infty \wedge a <_j 0))} \\
 & \texttt{\textbackslash then} & \multicolumn{4}{l}{(+\infty)} \\
 & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{3}{l}{(a = +\infty \vee a = -\infty \vee b = +\infty \vee b = -\infty)} \\
 & & \texttt{\textbackslash then} & \multicolumn{3}{l}{(-\infty)} \\
 & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{2}{l}{(((a = +0.0 \vee a = 0) \wedge (b = +0.0 \vee b = 0 \vee b >_j 0))} \\
 & & & & \multicolumn{2}{l}{\vee ((b = +0.0 \vee b = 0) \wedge (a = +0.0 \vee a = 0 \vee a >_j 0))} \\
 & & & & \multicolumn{2}{l}{\vee (a = -0.0 \wedge (b = -0.0 \vee b <_j 0))} \\
 & & & & \multicolumn{2}{l}{\vee (b = -0.0 \wedge (a = -0.0 \vee a <_j 0)))} \\
 & & & \texttt{\textbackslash then} & \multicolumn{2}{l}{(+0.0)} \\
 & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & (a = +0.0 \vee a = -0.0 \vee a = 0 \\
 & & & & & \vee b = +0.0 \vee b = -0.0 \vee b = 0)) \\
 & & & & \multicolumn{2}{l}{\texttt{\textbackslash then}(-0.0)} \\
 & & & & \multicolumn{2}{l}{\texttt{\textbackslash else}(a *_r b)))))}
\end{array}
}{a *_j b}
$$

and

$$
\frac{
\begin{array}{llllll}
\texttt{\textbackslash if} & \multicolumn{5}{l}{(a = NaN \vee b = NaN} \\
 & \multicolumn{5}{l}{\vee (a =_j 0 \wedge b =_j 0)} \\
 & \multicolumn{5}{l}{\vee (a = +\infty \wedge (b = +\infty \vee b = -\infty))} \\
 & \multicolumn{5}{l}{\vee (a = -\infty \wedge (b = +\infty \vee b = -\infty)))} \\
\texttt{\textbackslash then} & \multicolumn{5}{l}{(NaN)} \\
\texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{4}{l}{((a = +\infty \wedge b >_j 0) \vee (a = -\infty \wedge b <_j 0)} \\
 & & \multicolumn{4}{l}{\vee (a >_j 0 \wedge (b = +0.0 \vee b = 0))} \\
 & & \multicolumn{4}{l}{\vee (a <_j 0 \wedge b = -0.0))} \\
 & \texttt{\textbackslash then} & \multicolumn{4}{l}{(+\infty)} \\
 & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{3}{l}{(a = +\infty \vee a = -\infty \vee b =_j 0)} \\
 & & \texttt{\textbackslash then} & \multicolumn{3}{l}{(-\infty)} \\
 & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & \multicolumn{2}{l}{(((a = +0.0 \vee a = 0) \wedge b >_j 0)} \\
 & & & & \multicolumn{2}{l}{\vee (a = -0.0 \wedge (b <_j 0))} \\
 & & & & \multicolumn{2}{l}{\vee (a >_j 0 \wedge b = +\infty)} \\
 & & & & \multicolumn{2}{l}{\vee (a <_j 0 \wedge b = -\infty))} \\
 & & & \texttt{\textbackslash then} & \multicolumn{2}{l}{(+0.0)} \\
 & & & \texttt{\textbackslash else} & (\texttt{\textbackslash if} & (a =_j 0 \vee b = +\infty \vee b = -\infty) \\
 & & & & \multicolumn{2}{l}{\texttt{\textbackslash then}(-0.0)} \\
 & & & & \multicolumn{2}{l}{\texttt{\textbackslash else}(a /_r b)))))}
\end{array}
}{a /_j b}\ \textbf{exact_div}
$$

The remainder operator for exact floating-point arithmetic is rewritten by the following rule.

$$\frac{\begin{array}{lll} \texttt{\textbackslash if} & (a = NaN \lor b = NaN \lor a = +\infty \lor a = -\infty \lor b =_j 0) & \\ \texttt{\textbackslash then} & (NaN) & \\ \texttt{\textbackslash else} & (\texttt{\textbackslash if} \quad (b = +\infty \lor b = -\infty) & \\ & \texttt{\textbackslash then} \quad (a) & \\ & \texttt{\textbackslash else} \quad (\texttt{\textbackslash if} \quad (a = +0.0 \lor a = 0 & \\ & \qquad \qquad \lor(a >_j 0 \land \exists k \in \mathbb{Z} :\ a = b *_r k)) & \\ & \qquad \texttt{\textbackslash then} \quad (+0.0) & \\ & \qquad \texttt{\textbackslash else} \quad (\texttt{\textbackslash if} \quad (a = -0.0 & \\ & \qquad \qquad \qquad \lor(a <_j 0 \land \exists k \in \mathbb{Z} :\ a = b *_r k)) & \\ & \qquad \qquad \texttt{\textbackslash then} \quad (-0.0) & \\ & \qquad \qquad \texttt{\textbackslash else} \quad (a \%_r b)))) & \end{array}}{a \%_j b} \ \textbf{exact_mod}$$

Similar to the arithmetic operations, the predicates $=_j$, $\neq_j$, $<_j$, $\leq_j$, $>_j$ and $\geq_j$ for exact floating-point arithmetic are handled. Both, $=_j$ and $\neq_j$ are reduced to the equality symbol of the logic, whereas the others are rewritten by the corresponding relations of real arithmetic. Note, that the rewrite-rules given here do not apply on terms, but on formulae instead.

$$\frac{\begin{array}{ll} a \neq NaN \land b \neq NaN & \\ \land(& ((a = +0.0 \lor a = 0 \lor a = -0.0) \land (b = +0.0 \lor b = 0 \lor b = -0.0)) \\ & \lor\, a = b) \end{array}}{a =_j b} \ \textbf{exact_eq}$$

As $\neq_j$ is just the negation of $=_j$ the corresponding rewrite-rule is very simple:

$$\frac{\neg(a =_j b)}{a \neq_j b} \ \textbf{exact_neq}$$

The relations $<_j$ and $\leq_j$ are rewritten by:

$$\frac{\begin{array}{ll} a \neq NaN \land b \neq NaN \land a \neq +\infty \land b \neq -\infty \land a \neq_j b & \\ \land(& a = -\infty \lor b = +\infty \\ & \lor(a =_j 0 \land 0 <_r b) \lor (a <_r 0 \land b =_j 0) \\ & \lor a <_r b) \end{array}}{a <_j b} \ \textbf{exact_less}$$

and

$$\frac{a <_j b \vee a =_j b}{a \leq_j b} \textbf{ exact_leq}$$

And analogously, the relations $>_j$ and $\geq_j$ are rewritten by:

$$\frac{\begin{array}{l} a \neq NaN \wedge b \neq NaN \wedge a \neq -\infty \wedge b \neq +\infty \wedge a \neq_j b \\ \wedge(\quad a = +\infty \vee b = -\infty \\ \qquad \vee(a =_j 0 \wedge 0 >_r b) \vee (a >_r 0 \wedge b =_j 0) \\ \qquad \vee a >_r b) \end{array}}{a >_j b} \textbf{ exact_greater}$$

and

$$\frac{a >_j b \vee a =_j b}{a \geq_j b} \textbf{ exact_geq}$$

7 Conclusion

In this thesis the theory for an integration of basic floating-point arithmetic within the KeY system was developed. At first, a mathematical representation of floating-point numbers was introduced that is used within the logic JavaCardDL$^+$. The Java floating-point arithmetic was analyzed in chapter 3 and some non-intuitive peculiarities were shown in chapter 4.

In floating-point arithmetic, nearly all operations are inexact and suffer from rounding. To handle this in a sound way, the rounding function for the rounding mode `round-to-nearest` (which is Java's default rounding mode) was formalized in chapter 5. The fact, that rounding to the nearest can be described by the round-up and round-down functions eases the formalization.

By using the derived formalization a logic (JavaCardDL$^+$) and calculus for handling Java floating-point arithmetic has been developed in chapter 6. The logic is an extension of the existing logic JavaCardDL, which is used within the KeY system. In principle, the calculus and JavaCardDL$^+$ should make the application of formal methods to Java's floating-point arithmetic possible. In early stages of the software development process, algorithmic correctness is more important than the actual implementation. Therfore, a second semantics for JavaCardDL$^+$ has been developed that allows to ignore rounding issues and thus serves as a base for algorithmic verification.

7.1 Future work

In the scope of this thesis it is assumed that the underlying system is capable of handling real arithmetic. For the KeY tool this is not the case. However, it is possible to use external decision procedures within the tool. Therefore, the proof obligations (resulting from an incomplete proof) must be translated to the appropriate format and handed on to the decision procedure. There are several automated theorem provers, which are capable of (at least linear) real arithmetic such as the proof system Coq [19]. The translation of proof obligations should be rather straight forward and thus connecting the KeY tool with an external decision procedure would round off the handling of floating-point arithmetic.

Another way to achieve complete handling of floating-point arithmetic would be to develop and implement a calculus for handling real arithmetic within the KeY system itself. This could also be done only for exact floating-point arithmetic (arithmetic over $\widehat{\mathbb{F}}$, see section 2.2), as the arithmetic is based on a well defined mathemtical structure. Interesting would also be the reduction to integer arithmetic, which is possible because the arithmetic over $\widehat{\mathbb{F}}$ is based on integer arithmetic (see definition of operations on $\widehat{\mathbb{F}}$, definitions 2.2 and 2.7).

A Semantics of predefined rigid symbols

A.1 Symbols having the same semantics in all calculuses

The partial interpretation $\mathcal{I}_0$ of a JAVACARDDL$^+$ Kripke structure $\mathcal{K}$ (see definition 6.3) fixes the semantics of the following rigid symbols in both semantics SJava and SAlg.

$\mathcal{I}_0(\circ_r)(x,y)$	$= x \circ y$ (for $\circ \in \{+,-,*\}$)
$\mathcal{I}_0(/_r)(x,y)$	$= \begin{cases} x/y & , y \neq 0 \\ \textit{some arbitrary but fixed } d \in \mathcal{D}_0^{\texttt{real}} & \text{, otherwise} \end{cases}$
$\mathcal{I}_0(\%_r)(x,y)$	$= \begin{cases} r, \textit{ such that there is a } k \in \mathbb{Z} \textit{ with} & \\ x = k \cdot y + r \textit{ and } 0 \leq r < y & , y \neq 0 \\ \textit{some arbitrary but fixed } d \in \mathcal{D}_0^{\texttt{real}} & \text{, otherwise} \end{cases}$
$\mathcal{I}_0(\texttt{neg}_r)(x)$	$= -x$
$\mathcal{I}_0(r)$	$= r$ for each $r \in \mathbb{R}$
$\mathcal{I}_0(\texttt{NaN})$	$= NaN$
$\mathcal{I}_0(\texttt{+}\infty)$	$= +\infty$
$\mathcal{I}_0(\texttt{-}\infty)$	$= -\infty$
$\mathcal{I}_0(\texttt{+0.0})$	$= +0.0$
$\mathcal{I}_0(\texttt{-0.0})$	$= -0.0$
$\mathcal{I}_0(R)(m,e)$	$= m \cdot 2^e$
$\mathcal{I}_0(\texttt{precision}_{\texttt{jfloat}})$	$= 24$
$\mathcal{I}_0(\texttt{precision}_{\texttt{jdouble}})$	$= 53$
$\mathcal{I}_0(\texttt{maxMan}_{\texttt{jfloat}})$	$= 2^{24} - 1$
$\mathcal{I}_0(\texttt{maxMan}_{\texttt{jdouble}})$	$= 2^{53} - 1$
$\mathcal{I}_0(\texttt{maxExp}_{\texttt{jfloat}})$	$= 2^{7} - 1$
$\mathcal{I}_0(\texttt{maxExp}_{\texttt{jdouble}})$	$= 2^{11} - 1$
$\mathcal{I}_0(\texttt{minExp}_{\texttt{jfloat}})$	$= -2^{7} + 2$
$\mathcal{I}_0(\texttt{minExp}_{\texttt{jdouble}})$	$= -2^{11} + 2$
$\mathcal{I}_0(<_r)$	$= \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid x < y\}$
$\mathcal{I}_0(\leq_r)$	$= \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid x \leq y\}$
$\mathcal{I}_0(>_r)$	$= \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid x > y\}$
$\mathcal{I}_0(\geq_r)$	$= \{(x,y) \in \mathbb{R} \times \mathbb{R} \mid x \geq y\}$

$$\mathcal{I}_0(*_j)(x,y) = \begin{cases} NaN & \text{, if } x = NaN \vee y = NaN \\ & \vee(x = +\infty \wedge (y = +0.0 \vee y = -0.0 \vee y = 0)) \\ & \vee(x = -\infty \wedge (y = +0.0 \vee y = -0.0 \vee y = 0)) \\ & \vee(y = +\infty \wedge (x = +0.0 \vee x = -0.0 \vee x = 0)) \\ & \vee(y = -\infty \wedge (x = +0.0 \vee x = -0.0 \vee x = 0)) \\ +\infty & \text{, else if } (x = +\infty \wedge y >_j 0) \\ & \vee(x = -\infty \wedge y <_j 0) \\ & \vee(y = +\infty \wedge x <_j 0) \\ & \vee(y = -\infty \wedge x >_j 0) \\ -\infty & \text{, else if } (x = +\infty \wedge y <_j 0) \\ & \vee(x = -\infty \wedge y >_j 0) \\ & \vee(y = +\infty \wedge x >_j 0) \\ & \vee(y = -\infty \wedge x <_j 0) \\ +0.0 & \text{, else if } (x = 0 \wedge (y = +0.0 \vee y = 0 \vee y >_j 0)) \\ & \vee(x = +0.0 \wedge (y = +0.0 \vee y = 0 \vee y >_j 0)) \\ & \vee(x = -0.0 \wedge (y = -0.0 \vee y <_j 0)) \\ & \vee(x >_j 0 \wedge (y = +0.0 \vee y = 0)) \\ & \vee(x <_j 0 \wedge y = -0.0) \\ -0.0 & \text{, else if } (x = 0 \wedge (y = -0.0 \vee y <_j 0)) \\ & \vee(x = +0.0 \wedge (y = -0.0 \vee y <_j 0)) \\ & \vee(x = -0.0 \wedge (y = +0.0 \vee y = 0 \vee y >_j 0)) \\ & \vee(x >_j 0 \wedge y = -0.0) \\ & \vee(x <_j 0 \wedge (y = +0.0 \vee y = 0)) \\ \mathcal{I}_0(*_r)(x,y) & \text{, otherwise} \end{cases}$$

$$\mathcal{I}_0(\mathtt{neg}_j)(x) = \begin{cases} NaN & \text{, if } x = NaN \\ +\infty & \text{, if } x = -\infty \\ -\infty & \text{, if } x = +\infty \\ +0.0 & \text{, if } x = -0.0 \\ -0.0 & \text{, if } x = +0.0 \\ \mathcal{I}_0(\mathtt{neg}_r)(x) & \text{, otherwise} \end{cases}$$

$$\mathcal{I}_0(+_j)(x,y) = \begin{cases} NaN & \text{, if } x = NaN \vee y = NaN \\ & \vee(x = +\infty \wedge y = -\infty) \\ & \vee(x = -\infty \wedge y = +\infty) \\ +\infty & \text{, else if } x = +\infty \vee y = +\infty \\ -\infty & \text{, else if } x = -\infty \vee y = -\infty \\ -0.0 & \text{, else if } x = -0.0 \wedge y = -0.0 \\ +0.0 & \text{, else if } (x = -0.0 \wedge (y = +0.0 \vee y = 0)) \\ & \vee(x = +0.0 \wedge (y = +0.0 \vee y = -0.0 \vee y = 0)) \\ & \vee(x \in \mathbb{R} \wedge y \in \mathbb{R} \wedge x + y = 0) \\ y & \text{, else if } x =_j 0 \\ x & \text{, else if } y =_j 0 \\ \mathcal{I}_0(+_r)(x,y) & \text{, otherwise} \end{cases}$$

$$\mathcal{I}_0(-_j)(x,y) = \mathcal{I}_0(+_j)(x, \texttt{neg}_j(y))$$

$$\mathcal{I}_0(/_j)(x,y) = \begin{cases} NaN & \text{, if } x = NaN \vee y = NaN \\ & \vee(x = 0 \wedge (y = 0 \vee y = +0.0 \vee y = -0.0)) \\ & \vee(x = +0.0 \wedge (y = 0 \vee y = +0.0 \vee y = -0.0)) \\ & \vee(x = -0.0 \wedge (y = 0 \vee y = +0.0 \vee y = -0.0)) \\ & \vee(x = +\infty \wedge (y = +\infty \vee y = -\infty)) \\ & \vee(x = -\infty \wedge (y = +\infty \vee y = -\infty)) \\ +\infty & \text{, else if } (x = +\infty \wedge y >_j 0) \\ & \vee(x = -\infty \wedge y <_j 0) \\ & \vee((y = +0.0 \vee y = 0) \wedge x >_j 0) \\ & \vee(y = -0.0 \wedge x <_j 0) \\ -\infty & \text{, else if } (x = +\infty \wedge y <_j 0) \\ & \vee(x = -\infty \wedge y >_j 0) \\ & \vee((y = +0.0 \vee y = 0) \wedge x <_j 0) \\ & \vee(y = -0.0 \wedge x >_j 0) \\ +0.0 & \text{, else if } ((x = +0.0 \vee x = 0) \wedge y >_j 0) \\ & \vee(x = -0.0 \wedge y <_j 0) \\ & \vee(y = +\infty \wedge x >_j 0) \\ & \vee(y = -\infty \wedge x <_j 0) \\ -0.0 & \text{, else if } ((x = +0.0 \vee x = 0) \wedge y <_j 0) \\ & \vee(x = -0.0 \wedge y >_j 0) \\ & \vee(y = +\infty \wedge x <_j 0) \\ & \vee(y = -\infty \wedge x >_j 0) \\ \mathcal{I}_0(/_r)(x,y) & \text{, otherwise} \end{cases}$$

$$\mathcal{I}_0(\%_j)(x,y) = \begin{cases} NaN & \text{, if } x = NaN \vee y = NaN \\ & \vee x = +\infty \vee x = -\infty \\ & \vee y = +0.0 \vee y = -0.0 \vee y = 0 \\ x & \text{, else if } y = +\infty \vee y = -\infty \\ +0.0 & \text{, else if } x = +0.0 \vee x = 0 \\ & \vee (x >_j 0 \wedge \exists k \in \mathbb{Z} : x = y *_r k) \\ -0.0 & \text{, else if } x = -0.0 \\ & \vee (x <_j 0 \wedge \exists k \in \mathbb{Z} : x = y *_r k) \\ \mathcal{I}_0(\%_r)(x,y) & \text{, otherwise} \end{cases}$$

In the following let $\mathbb{R}^+$ be defined as $\mathbb{R}^+ := \mathbb{R} \cup \{+0.0, -0.0, +\infty, -\infty\}$

$$\begin{aligned}
\mathcal{I}_0(=_j) &= \{(x,y) \in \mathbb{R}^+ \times \mathbb{R}^+ |\ x = y\} \cup \{(0,+0.0),(0,-0.0)\} \\
&\quad \cup \{(+0.0,0),(+0.0,-0.0),(-0.0,0),(-0.0,+0.0)\} \\
\mathcal{I}_0(\neq_j) &= (\mathbb{R}^+ \cup \{NaN\}) \times (\mathbb{R}^+ \cup \{NaN\}) \setminus \mathcal{I}_0(=_j) \\
\mathcal{I}_0(<_j) &= \{(x,y) \in \mathbb{R} \times \mathbb{R} |\ x < y\} \\
&\quad \cup \{(x,+\infty) |\ x \in \mathbb{R} \cup \{+0.0,-0.0\}\} \\
&\quad \cup \{-\infty, y) |\ y \in \mathbb{R} \cup \{+0.0,-0.0\}\} \\
&\quad \cup \{(+0.0,y),(-0.0,y) |\ y \in \mathbb{R} \wedge y > 0\} \\
&\quad \cup \{(x,+0.0),(x,-0.0) |\ x \in \mathbb{R} \wedge x < 0\} \\
&\quad \cup \{(-\infty,+\infty)\} \\
\mathcal{I}_0(\leq_j) &= \mathcal{I}_0(<_j) \cup \mathcal{I}_0(=_j) \\
\mathcal{I}_0(>_j) &= \{(x,y) \in \mathbb{R} \times \mathbb{R} |\ x > y\} \\
&\quad \cup \{(+\infty,y) |\ y \in \mathbb{R} \cup \{+0.0,-0.0\}\} \\
&\quad \cup \{x,-\infty) |\ x \in \mathbb{R} \cup \{+0.0,-0.0\}\} \\
&\quad \cup \{(+0.0,y),(-0.0,y) |\ y \in \mathbb{R} \wedge y < 0\} \\
&\quad \cup \{(x,+0.0),(x,-0.0) |\ x \in \mathbb{R} \wedge x > 0\} \\
&\quad \cup \{(+\infty,-\infty)\} \\
\mathcal{I}_0(\geq_j) &= \mathcal{I}_0(>_j) \cup \mathcal{I}_0(=_j)
\end{aligned}$$

A.2 Semantics reflecting Java's behavior

This section fixes the interpretation of rigid symbols for the semantics $\mathcal{S}_{\text{JAVA}}$.

To improve the readability of the following, two syntactic predicates *isNormalized* and *isDenormalized* are introduced.

Definition A.1
Let $T \in \{\mathtt{jfloat}, \mathtt{jdouble}\}$ *and* $x \in \widehat{\mathbb{F}}$. *The syntactic predicates* $isNormalized_T(\widehat{\mathbb{F}})$

and $isDenormalized_T(\widehat{\mathbb{F}})$ *are defined by*

$$\begin{aligned} isNormalized_T(x) &:= \exists m, e \in \mathbb{Z}: \frac{\mathit{maxMan}_T + 1}{2} \leq |m| \leq \mathit{maxMan}_T \\ &\quad \wedge \mathit{minExp}_T \leq e + \mathit{precision}_T - 1 \leq \mathit{maxExp}_T \\ &\quad \wedge x = m \cdot 2^e \\ isDenormalized_T(x) &:= \exists m \in \mathbb{Z}: 0 \leq |m| < \frac{\mathit{maxMan}_T + 1}{2} \\ &\quad \wedge x = m \cdot 2^{\mathit{minExp}_T - \mathit{precision}_T + 1} \end{aligned}$$

where $\mathit{precision}_T$, minExp_T *and* maxExp_T *are syntactic sugar for the format specific values defined above as interpretation of the identically named function symbols.*

The interpretation within the semantics $\mathcal{S}_{\text{JAVA}}$ is fixed by the following definitions.

$$\mathcal{I}_0(\mathtt{inFloat}) = \mathcal{D}_0^{\mathtt{SVDomain}} \cup \{x \in \widehat{\mathbb{F}} |\ isNormalized_{\mathtt{jfloat}}(x) \vee isDenormalized_{\mathtt{jfloat}}(x)\}$$

$$\mathcal{I}_0(\mathtt{inDouble}) = \mathcal{D}_0^{\mathtt{SVDomain}} \cup \{x \in \widehat{\mathbb{F}} |\ isNormalized_{\mathtt{jdouble}}(x) \vee isDenormalized_{\mathtt{jdouble}}(x)\}$$

$$\mathcal{I}_0(\mathtt{round}_{\mathtt{jfloat}})(x) = \begin{cases} x & \text{, if } x \in \mathcal{D}_0^{\mathtt{SVDomain}} \\ +\infty & \text{, else if } x \geq 2^{\mathtt{maxExp}_{\mathtt{jfloat}}}(2 - 2^{-\mathtt{precision}_{\mathtt{jfloat}}}) \\ -\infty & \text{, else if } x \leq -2^{\mathtt{maxExp}_{\mathtt{jfloat}}}(2 - 2^{-\mathtt{precision}_{\mathtt{jfloat}}}) \\ +0.0 & \text{, else if } x \geq 0 \wedge x \leq 2^{\mathtt{minExp} - \mathtt{precision}_{\mathtt{jfloat}}} \\ -0.0 & \text{, else if } x < 0 \wedge x \geq -2^{\mathtt{minExp} - \mathtt{precision}_{\mathtt{jfloat}}} \\ d & \text{, otherwise} \end{cases}$$

where $d \in \widehat{\mathbb{F}}$ satisfies the formula

$$\begin{aligned} & \mathtt{inFloat}(d) \\ \wedge & \left(\forall c \in \widehat{\mathbb{F}}: \mathtt{inFloat}(c) \rightarrow abs(d - x) \leq abs(c - x)\right) \\ \wedge & \Big(\exists a \in \widehat{\mathbb{F}}\ \exists b \in \widehat{\mathbb{F}}: \quad \mathtt{inFloat}(a) \wedge \mathtt{inFloat}(b) \wedge a \neq b \\ & \qquad \wedge \left(\forall c \in \widehat{\mathbb{F}}: \mathtt{inFloat}(c) \rightarrow abs(a - x) \leq abs(c - x)\right) \\ & \qquad \wedge \left(\forall c \in \widehat{\mathbb{F}}: \mathtt{inFloat}(c) \rightarrow abs(b - x) \leq abs(c - x)\right) \\ & \quad \rightarrow \left[\mathtt{maxMan}_{\mathtt{jfloat}} / \tfrac{\mathtt{maxMan}_{\mathtt{jfloat}} + 1}{2}\right] \mathtt{inFloat}(\tfrac{d}{2}) \Big) \end{aligned}$$

$abs(x)$ denotes the absolute value of $x \in \mathbb{R}$ and $[x/y]$ denotes the substitution of x by y.

The interpretation of $\mathtt{round}_{\mathtt{jdouble}}$ is fixed analogously by replacing `jfloat` with `jdouble` and `inFloat` with `inDouble`.

$$\mathcal{I}_0(\mathtt{addJ}_T)(x, y) = \mathcal{I}_0(\mathtt{round}_T)(\mathcal{I}_0(+_j)(x, y))$$

$\mathcal{I}_0(\texttt{subJ}_T)(x,y)$	$= \mathcal{I}_0(\texttt{round}_T)(\mathcal{I}_0(-_j)(x,y))$
$\mathcal{I}_0(\texttt{mulJ}_T)(x,y)$	$= \mathcal{I}_0(\texttt{round}_T)(\mathcal{I}_0(*_j)(x,y))$
$\mathcal{I}_0(\texttt{divJ}_T)(x,y)$	$= \mathcal{I}_0(\texttt{round}_T)(\mathcal{I}_0(/_j)(x,y))$
$\mathcal{I}_0(\texttt{modJ}_T)(x,y)$	$= \mathcal{I}_0(\%_j)(x,y)$
$\mathcal{I}_0(\texttt{negJ}_T)(x)$	$= \mathcal{I}_0(\texttt{neg}_j)(x)$
$\mathcal{I}_0(\texttt{castToJ}_T)(x)$	$= \mathcal{I}_0(\texttt{round}_T)(x)$

The interpretation of the rounding functions $\Box_T$, $\triangle_T$ and $\bigtriangledown_T$ are fixed such that the mathematical functions $\Box_{R_C}$, $\triangle_{R_C}$ and $\bigtriangledown_{R_C}$ of section 5.4 are modelled.

A.3 Semantics for algorithmic verification

This section fixes the interpretation of rigid symbols for the semantics $\mathcal{S}_{\texttt{ALG}}$. As rounding is ignored here, the corresponding functions will be underspecified. And thus, no interpretation will be given for them. The JAVA operations are handled as if they were exact floating-point operations.

$\mathcal{I}_0(\texttt{addJ}_T)(x,y)$	$= \mathcal{I}_0(+_j)(x,y)$
$\mathcal{I}_0(\texttt{subJ}_T)(x,y)$	$= \mathcal{I}_0(-_j)(x,y)$
$\mathcal{I}_0(\texttt{mulJ}_T)(x,y)$	$= \mathcal{I}_0(*_j)(x,y)$
$\mathcal{I}_0(\texttt{divJ}_T)(x,y)$	$= \mathcal{I}_0(/_j)(x,y)$
$\mathcal{I}_0(\texttt{modJ}_T)(x,y)$	$= \mathcal{I}_0(\%_j)(x,y)$
$\mathcal{I}_0(\texttt{negJ}_T)(x)$	$= \mathcal{I}_0(\texttt{neg}_j)(x)$
$\mathcal{I}_0(\texttt{castToJ}_T)(x)$	$= x$

Bibliography

[1] Advanced Micro Devices™ inc. http://www.amd.com.

[2] Eclipse Foundation. http://www.eclipse.org.

[3] Intel™ corp. http://www.intel.com.

[4] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[5] M. Baum. Debugging by Visualizing Symbolic Execution. Diploma thesis, Universität Karlsruhe, July 2007.

[6] B. Beckert, R. Hähnle, and P. S. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334/2007 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[7] S. Boldo and J.-C. Filliâtre. Formal verification of floating-point programs. In *IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE Computer Society, 2007.

[8] M. Daumas, L. Rideau, and L. Théry. A generic library for floating-point numbers and its application to exact computing. In *TPHOLs '01: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, London, UK, 2001. Springer-Verlag.

[9] J.-C. Filliâtre and C. Marché. Multi-prover Verification of C Programs. *Lecture Notes in Computer Science*, 3308:15–29, 2004.

[10] J. Forum. Java Grande Forum Report: Making Java Work for High-End Computing, 1998.

[11] Free Software Foundation. Documentation of gcj, Guide to GNU gcj. available online: http://gcc.gnu.org/onlinedocs/gcj/.

[12] C. Gladisch. Verification of C with KeY. Diploma thesis, Universität Koblenz, March 2006.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[16] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[17] J. Harrison. Floating point verification in HOL Light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/tang.html`.

[18] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[19] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant - A tutorial. technical report 178, INRIA, July 1995.

[20] IEEE. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, Feb. 1985.

[21] W. Kahan. How Java's Floating-Point Hurts Everyone Everywhere. Presented at ACM 1998 Workshop on Java for High-Performance Network Computing, March 1998.

[22] M. Kaufmann, P. Manolios, and S. Moore. *Computer-aided reasoning: An Approach*. Springer Verlag, Norwell, MA, USA, 2000.

[23] U. Kulisch. *Grundlagen des Numerischen Rechnens – Mathematische Begründung der Rechnerarithmetik*. Reihe Informatik, Band 19. Mannheim/Wien/Zürich, 1976.

[24] U. Kulisch and W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, New York, 1981.

[25] P. S. Miner. Defining the ieee-854 floating-point standard in pvs. Technical report, NASA, 1995.

[26] D. Monniaux. The pitfalls of verifying floating-point computations. 2007.

[27] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 411–414, New Brunswick, NJ, USA, 1996. Springer Verlag.

[28] D. M. Russinoff. A mechanically checked proof of ieee compliance of the floating point multiplication, division and square root algorithms of the amd-k7 tm processor. In *LMS Journal of Computation and Mathematics*, volume 1, pages 148–200, December 1998.

[29] S. Schlager. Handling of Integer Arithmetic in the Verification of Java Programs. Diploma-Thesis, Universität Karlsruhe, Karlsruhe, May 2002.

[30] P. H. Schmitt, B. Beckert, and R. Hähnle. The KeY-project http://www.key-project.org, 1998-.

[31] J. M. Spivey. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, new york, usa edition, 1988.