Formalizing Java's Two's-Complement Integral Type in Isabelle/HOL
ETH Technical Report 458

Nicole Rauch and Burkhart Wolff

November 16, 2004

Abstract

We present a formal model of the Java two's-complement integral arithmetics. The model directly formalizes the arithmetic operations as given in the Java Language Specification (JLS). The algebraic properties of these definitions are derived. Underspecifications and ambiguities in the JLS are pointed out and clarified. The theory is formally analyzed in Isabelle/HOL, that is, machine-checked proofs for the ring properties and divisor/remainder theorems etc. are provided. This work is suited to build the framework for machine-supported reasoning over arithmetic formulae in the context of Java source-code verification.

Contents

1 Introduction **5**
1.1 Related Work . . . 6
1.2 Outline of this Report . . . 8

2 Overview of the Theories **9**

3 Basic Type Construction via Strictification, Smashing and Lifting **10**
3.1 A Generic Theory of Undefinedness, Strictness, Smashing . . 11
3.2 A Theory of Undefinedness and Strictness in Lifted Types . . 12
3.3 A Generic Theory of Strictness . . . 13

4 Formalizing the Normal Behavior Java Integers **14**
4.1 General Definitions . . . 15
4.1.1 General Lemmas . . . 17
4.2 Axclass zero . . . 21
4.2.1 zero Lemmas . . . 22
4.3 Axclass one . . . 22
4.3.1 one Lemmas . . . 22
4.4 Axclass number . . . 22

5 Addition and Subtraction **23**
5.1 Axclass plus . . . 23
5.1.1 plus Lemmas . . . 23
5.2 Axclass minus . . . 26
5.2.1 minus Lemmas . . . 27
5.3 Axclass plus_ac0 . . . 30

6 Multiplication **31**
6.1 Axclass times . . . 31
6.1.1 times Lemmas . . . 31

7 The algebraic hierarchy of rings as axiomatic classes **36**

8 Constants **36**

9 Axioms **37**
9.1 Ring axioms . . . 37
9.2 Integral domains . . . 37
9.3 Factorial domains . . . 37
9.4 Euclidean domains . . . 38
9.5 Fields . . . 38

10 Basic facts **38**
10.1 Normaliser for rings . . . 38
10.2 Rings and the summation operator . . . 38
10.3 Integral Domains . . . 40

11 Auxiliary Arithmetic Lemmas **40**

12 Ordering Properties **42**
12.1 Axclass ord . . . 42
12.1.1 ord Lemmas . . . 43
12.2 Axclass order . . . 44
12.3 Axclass linorder . . . 44
12.3.1 linorder Lemmas . . . 45
12.4 The Absolute Value Function `abs` . . . 50
12.4.1 `abs` Lemmas . . . 51
12.5 Formalizing More plus Lemmas . . . 52

13 Ring Properties **56**
13.1 Axclass power . . . 56
13.2 Axclass inverse . . . 56
13.3 Axclass ring . . . 56

14 Division Operator **57**
14.1 Division Definition . . . 57
14.2 Division Representation Boundedness . . . 58
14.3 Division Characterizations . . . 61

15 Remainder Operator **62**
15.1 Remainder Definition . . . 62
15.2 Remainder Representation Boundedness . . . 63
15.3 Remainder Characterizations . . . 68

16 Calculating Example Values for div and mod **70**
16.1 Division and Modulo Lemmas: Base Cases . . . 70
16.2 Sign of Divisions . . . 76
16.3 Sign of Remainders . . . 77
16.4 Bounds of the Division Operation . . . 85
16.5 Bounds on the Remainder . . . 94
16.6 Main Theorem: The div/mod lemma . . . 102
16.7 Calculating Example Values for div and mod . . . 114

17 Formalization With Bitstring Representation **118**
17.1 Bitwise Operators - AND, OR, XOR . . . 121

18 Formalizing the Exceptional Behavior Java Integers **123**
18.1 The One-Exception Theory Morphism 124
18.2 Proof of Basic Definedness 126
18.3 Proof of exception propagation rules 126
18.4 Lifting Axiomatic Classes to Versions with Exceptional Behaviour 127
18.5 Lifting Number Representations and Bridging to Computations 130
18.6 Lifting JavaInteger Theorems 131
18.7 The Multiple-Exception Theory Morphism 133
18.8 Lifting orders and related theorems. 137

19 Conclusions and Future Work **140**

1 Introduction

Admittedly, modelling numbers in a theorem prover is not really a "sexy subject" at first sight. Numbers are fundamental, well-studied and well-understood, and everyone is used to them since school-mathematics. Basic theories for the naturals, the integers and real numbers are available in all major theorem proving systems (e.g. [GM93a, The02, Pau94]), so why care?

However, numbers as specified in a concrete processor or in a concrete programming language semantics are oriented towards an efficient implementation on a machine. They are finite datatypes and typically based on bit-fiddling definitions. Nevertheless, they often possess a surprisingly rich theory (ring properties, for example) that also comprises a number of highly non-standard and tricky laws with non-intuitive and subtle preconditions.

In the context of program verification tools (such as the B tool [Abr96], KIV [BRS+00], LOOP [BJ01], and Jive [MPH00], which directly motivated this work), efficient numerical programs, e.g. square roots, trigonometric functions, fast Fourier transformation or efficient cryptographic algorithms represent a particular challenge. Fortunately, theorem proving technology has matured to a degree that the analysis of realistic machine number specifications for widely-used programming languages such as Java or C now is a routine task [Har99].

With respect to the formalization of integers, we distinguish two approaches:

(1) the *partial approach*: the arithmetic operations + − * / % are only defined on an interval of (mathematical) integers, and left undefined whenever the result of the operation is outside the interval (c.f. [BS02], which is mainly geared towards this approach).

(2) the *wrap-around approach*: integers are defined on $[-2^{n-1} .. 2^{n-1}-1]$, where in case of overflow the results of the arithmetic operations are mapped back into this interval through modulo calculations. These numbers can be equivalently realized by bitstrings of length n in the widely-used two's-complement representation system [Gol02].

While in the formal methods community there is a widespread reluctance to integrate machine number models and therefore a tendency towards either (infinite) mathematical integers or the first approach ("either remain fully formal but focus on a smaller or simpler language [...]; or remain with the real language, but give up trying to achieve full formality." [ST99]), we strongly argue in favour of the second approach for the following reasons:

(1) In a wrap-around implementation, certain properties like "Maxint + 1 = Minint" hold. This has the consequence that crucial algebraic properties such as the associativity law "a + (b + c) = (a + b) + c"

hold in the wrap-around approach, but not in the partial approach. The wrap-around approach is therefore more suited for automated reasoning.

(2) Simply using the mathematical operators on a subset of the mathematical integers does not handle surprising definitions of operators appropriately. E.g. in Java the result of an integer division is always rounded towards zero, and thus the corresponding modulo operation can return negative values. This is unusual in mathematics. Therefore, this naïve approach does not only disregard overflows and underflows but also disregards unconventionally defined operators.

(3) The Java type `int` is defined in terms of wrap-around in the Java Language Specification [GJSB00], so why should a programmer who strictly complies to it in an efficient program be punished by the lack of formal analysis tools?

(4) Many parts of the JLS have been analyzed formally — so why not the part concerning number representations? There are also definitions and claimed properties that should be checked; and there are also possible inconsistencies or underspecifications as in all other informal specifications.

As technical framework for our analysis we use Isabelle/HOL and the Isar proof documentation package, whose output is directly used throughout this report, including all proofs. Isabelle [Pau94] is a *generic* theorem prover, i.e. new object logics can be introduced by specifying their syntax and inference rules. Isabelle/HOL is an instance of Isabelle with Church's *higher-order logic* (HOL) [GM93b], a classical logic with equality enriched by total polymorphic higher-order functions. In HOL, induction schemes can be expressed inside the logic, as well as (total) functional programs. Isabelle's methodology for safely building up large specifications is the decomposition of problems into *conservative extensions*. A conservative extension introduces new constants (by *constant definitions*) and types (by *type definitions*) only via axioms of a particular, machine-checked form; a proof that conservative extensions preserve consistency can be found in [GM93b]. Among others, the HOL library provides conservative theories for the logical type *bool*, for the numbers such as *int* and for bitstrings *bin*.

1.1 Related Work

The formalization of IEEE floating point arithmetics has attracted the interest of researchers for some time [CnM95, AS95], e.g. leading to concrete, industry strength verification technologies used in Intel's IA64 architecture [Har99].

In hardware verification, it is a routine task to verify two's complement number operations and their implementations on the gate level. Usually, variants of *binary decision diagrams* are used to represent functions over bit words canonically; thus, if a trusted function representation is identical to one generated from a highly complex implementation, the latter is verified. Meanwhile, addition, multiplication and restricted forms of operations involving division and remainder have been developed [HD99]. Unfortunately, it is known that one variant particularly suited for one operation is inherently intractable for another, etc. Moreover, general division and remainder functions have been proven to be intractable by word-level decision diagrams (WLDD) [SBW02]. For all these reasons, the approach is unsuited to investigate the *theory* of two's complement numbers: for example, the theorem JavaInt-div-mod (see Sec. 15), which involves a mixture of all four operations, can only be proven up to a length of 9 bits, even with leading edge technology WLDD packages[1].

Amazingly, *formalized theories* of the two's complement number have only been considered recently; i.e. Fox formalized 32-bit words and the ARM processor for HOL [Fox01], and Bondyfalat developed a (quite rudimentary) bit words theory with division in the AOC project [Bon]. In the context of Java and the JLS, Jacobs [Jac03] presented a fragment of the theory of integral types. This work (like ours) applies to Java Card as well since the models of the four smaller integral types (excluding long) of Java and Java Card are identical [Sun00, § 2.2.3.1]. However, although our work is in spirit and scope very similar to [Jac03], there are also significant differences:

- We use standard integer intervals as reference model for the arithmetic operations as well as two's-complement bitstrings for the bitshift and the bitwise AND, OR, XOR operations (which have not been covered by [Jac03] anyway). Where required, we prove lemmas that show the isomorphy between these two representations.

- While [Jac03] just presents the normal behavior of arithmetic expressions, we also cover the exceptional behavior for expressions like "x / 0" by adding a second theory layer with so-called strictness principles (see Sec. 18).

- [Jac03] puts a strong emphasis on widening and narrowing operations which are required for the Java types short and byte. We currently only concentrate on the type int and therefore did not model widening and narrowing yet.

The Java Virtual Machine (JVM) [LY96] has been extensively modelled in the Project Bali [Pus98]. However, the arithmetic operations in this JVM

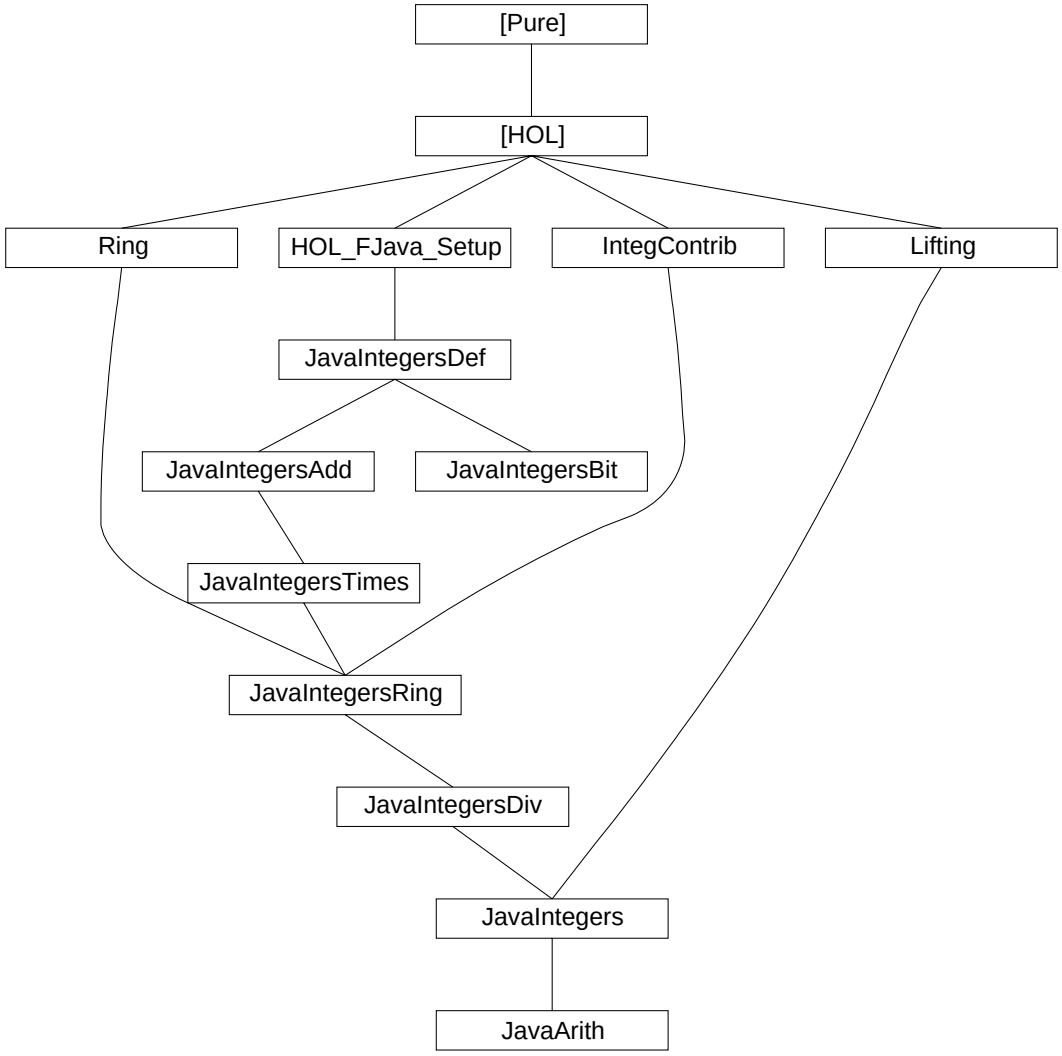
[1] Thanks to Marc Herbstritt [Her03] to check this for us!

model are based on mathematical integers. Since our work is based on the same system, namely Isabelle2002, our model of a two's-complement integer datatype could replace the mathematical integers in this JVM theory.

1.2 Outline of this Report

The specification is divided into two parts: the normal behaviour Java integers (type $JavaInt$) and the exceptional behaviour Java integers (type $java\_int$). In Sec. 2 a graphical overview of the theories is given. After an initial Sec. 3 containing elementary semantic constructions, the first part starts with Sec. 4, which introduces the core conservative definitions of the data type as interval of the "mathematical" integral numbers. Sec. 5 formalizes the addition and Sec. 6 treats the multiplication. Some auxiliary lemmas which belong to Isabelle/HOL's int theory are given in Sec. 11. Ordering properties of $JavaInt$ are given in Sec. 12. The ring properties are analysed in Sec. 13. Part one continues with Sec. 14, Sec. 15 and Sec. 16 which present the division and remainder theories and prove some example calculations. Sec. 17 gives a rudimentary theory of bitwise operations based on the library *bits* theory (this section is merely a proof of concept and not further used in this work). Part two is concerned with the exceptional behaviour of Java operators and consists of Sec. 18.

2 Overview of the Theories



theory *Lifting* = *Main* :

3 Basic Type Construction via Strictification, Smashing and Lifting

The main purpose of this theory is to provide a generic theory of undefinedness. The Isabelle standard mechanism for such a generic data type is the class mechanism. The standard method is to declare a class to which all related operations are associated.

classes $bot < type$

consts $UU :: 'a::bot$

constdefs $DEF :: 'a::bot \Rightarrow bool$
$DEF\ x \equiv (x \neq UU)$
$is\text{-}strict :: ('a::bot \Rightarrow 'b::bot) \Rightarrow bool$

$is\text{-}strict\ f \equiv (f\ UU = UU)$

$strictify :: (('a::bot) \Rightarrow ('b::bot)) \Rightarrow 'a \Rightarrow 'b$
$strictify\ f\ x \equiv if\ x{=}UU\ then\ UU\ else\ f\ x$

$smash :: [['b::bot,\ 'a::bot] \Rightarrow bool,\ 'a] \Rightarrow 'a$
$smash\ f\ X \equiv if\ f\ UU\ X\ then\ UU\ else\ X$

We introduce now the lifting construction (Winskel, pp.132) by a type constructor defined as free data type:

datatype $'a\ up = lift\ 'a \mid down$

constdefs $drop :: 'a\ up \Rightarrow 'a$
$drop\ x \equiv case\ x\ of\ lift\ v \Rightarrow v \mid down \Rightarrow @x.\ True$

syntax $@lift :: 'a \Rightarrow 'a\ up\ \ (\lfloor(\text{-})\rfloor\ 10)$
$@drop :: 'a\ up \Rightarrow 'a\ \ (\lceil(\text{-})\rceil\ 10)$

translations
$\lfloor a \rfloor == lift\ a$
$\lceil a \rceil == drop\ a$

The class is then propagated across lifting, function space and cartesian products.

arities $up :: (type)\ bot$
arities $fun :: (type,bot)\ bot$
arities $* :: (bot,bot)\ bot$

defs (**overloaded**)
$UU\text{-}up\text{-}def : UU \equiv down$

UU-fun-def : $UU \equiv (\lambda x.\ UU)$
UU-pair-def : $UU \equiv (UU,\ UU)$

instance *up* :: (*type*) *bot* **..**
instance *fun* :: (*type*,*bot*) *bot* **..**
instance $*$ :: (*bot*,*bot*) *bot* **..**

declare *DEF-def* [*simp*] *UU-up-def* [*simp*] *UU-fun-def* [*simp*]

3.1 A Generic Theory of Undefinedness, Strictness, Smashing

lemma *not-DEF-UU* [*simp*]: $\neg$ *DEF*(*UU*)
by (*simp* (*no-asm*))

lemma *is-strict-strictify*:
is-strict(*strictify f*)
by (*auto simp*: *is-strict-def strictify-def*)

lemma *strict2a-UU* [*simp*] :
strictify f UU = *UU*
by (*simp* (*no-asm*) *add*: *strictify-def*)

lemma *strict2b-UU* [*simp*] :
strictify f UU X = *UU*
by (*simp* (*no-asm*) *add*: *strictify-def*)

lemma *DEF-strictify-DEF-args2* :
$!!f.$ *DEF*(*strictify* ($\lambda x.$ *strictify*(*f x*)) *X Y*) $\Longrightarrow$ *DEF X* & *DEF Y*
apply (*simp add*: *strictify-def*)
apply(*case-tac X*=*UU*)
apply *auto*
done

lemma *DEF-strictify-DEF-fun* :
$!!f.$ $[\![$ $!!X.$ *DEF*(*f X*) ; *DEF X* $]\!]$ $\Longrightarrow$ *DEF*(*strictify f X*)
by (*auto simp*: *strictify-def*)

lemma *DEF-strictify-DEF-args*:
$!!f.$ *DEF*(*strictify f X*) $\Longrightarrow$ *DEF f* & *DEF X*
by (*auto simp*: *strictify-def*)

lemma *is-strict-compose* :
$!!\ f.$ $[\![$ *is-strict f*; *is-strict g* $]\!]$ $\Longrightarrow$ *is-strict* (*f o g*)
by (*auto simp*: *is-strict-def o-def*)

lemma *smash-strict* [*simp*] :

smash f UU = *UU*
by (*simp* (*no-asm*) *add*: *smash-def*)

lemma *smashed-sets-nonempty* [*intro*!] :
$UU \in \{X.\ smash\ f\ X = X\}$
by (*simp* (*no-asm*))
— for proofs of non-emptiness of type-definition based on smashed collection types

3.2 A Theory of Undefinedness and Strictness in Lifted Types

lemma *not-down-exists-lift* :
$x \neq down = (\exists\ y.\ x = (\lfloor y \rfloor))$
by (*induct-tac x*, *auto*)

lemma *not-down-exists-lift2* :
$x \neq UU = (\exists\ y.\ x = (\lfloor y \rfloor))$
by (*induct-tac x*, *auto*)

lemma *drop-lift* [*simp*]:
$(\lceil\ \lfloor f \rfloor\ \rceil) = f$
by (*auto simp*: *drop-def*)

lemma *drop-down* [*simp*]:
$(\lceil UU \rceil) = (@\ x.\ True)$
by (*auto simp*: *drop-def*)

lemma *not-DEF-down* [*simp*]:
$\neg\ DEF(down)$
by (*simp* (*no-asm*))

lemma *DEF-lift* [*simp*]:
$DEF(\lfloor x \rfloor)$
by (*simp* (*no-asm*))

lemma *DEF-X-up* :
$DEF(X::'a\ up) = (\exists\ x.\ X = (\lfloor x \rfloor))$
by (*simp add*: *not-down-exists-lift*)

lemma *not-DEF-X-up* [*simp*]:
$(\neg\ DEF(X::'a\ up)) = (X = UU)$
by (*simp* (*no-asm*))

lemma *DEF-fun-lift* [*simp*]:
$DEF(\lambda x.\ \lfloor (f\ x) \rfloor)$
apply(*simp* (*no-asm*))
apply (*rule notI*)
apply (*drule fun-cong*)
apply *simp*
done

lemma *DEF-fun-fun-lift* [*simp*]:
$DEF(\lambda x\ y.\ \lfloor(f\ x\ y)\rfloor)$
apply(*simp* (*no-asm*))
apply (*rule notI*)
apply (*drule fun-cong*) +
apply *simp*
done

lemma *drop-lift-idem*:
$(\lceil\lfloor x\rfloor\rceil) = x$
by (*simp split add*: *up.split*)

lemma *lift-drop-idem*:
$[\![\ DEF(x)\ ]\!] \Longrightarrow (\lfloor\lceil x\rceil\rfloor) = x$
  apply *simp*
  apply (*cases x*)
  apply (*simp-all add*: *up.inject*)
  done

lemma *lift-defined*:
  $UU \notin X \Longrightarrow (\lfloor X\rfloor) \neq UU$
by (*auto*)

3.3 A Generic Theory of Strictness

lemma *strict2a2-UU* [*simp*]:
strictify f down = *down*
by (*simp* (*no-asm*) *add*: *strictify-def*)

lemma *strict2c-UU* [*simp*]:
strictify f down X = *down*
by (*simp* (*no-asm*) *add*: *strictify-def*)

lemma *strict2d-UU* [*simp*]:
strictify f UU X = *UU*
by (*simp* (*no-asm*) *add*: *strictify-def*)

lemma *strict2e-UU* [*simp*]:
strictify (λx. *strictify* ($f\ x$)) (X::$'a$::*bot*) *UU* = *UU*
by (*simp add*: *strictify-def split*: *split-if*)

lemma *strict2f-UU* [*simp*]:
strictify (λx. *strictify* ($f\ x$)) (X::$'a$::*bot*) *down* = *down*
by (*simp add*: *strictify-def split*: *split-if*)

lemma *strict2-DEF* [*simp*]:
$!!X.\ DEF\ X \Longrightarrow strictify\ f\ X = f\ X$
by (*auto simp*: *strictify-def*)

lemma *strict22-DEF* [*simp*]:
$!!X.\ DEF\ X \Longrightarrow DEF\ Y \Longrightarrow$
$strictify\ (\lambda x.\ strictify\ (f\ x))\ (X::'a::bot)\ Y = f\ X\ Y$
by(*simp add*: *DEF-X-up strictify-def split*: *split-if*)

declare *UU-up-def* [*simp del*] *DEF-def* [*simp del*]

end

4 Formalizing the Normal Behavior Java Integers

theory *JavaIntegersDef* = *HOL-FJava-Setup* :

The formalization of Java integers models the primitive Java type `int` as closely as possible. The programming language Java comes with a quite extensive language specification [GJSB00] which tries to be accurate and detailed. Nonetheless, there are several white spots in the Java integer specification which are pointed out in this report. The language Java itself is platform-independent. The bit length of the data type `int` is fixed in a machine-independent way. This simplifies the modelling task. The JLS states about the integral types:

Java Language Specification [GJSB00], §4.2

> "The integral types are byte, short, int, and long, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and char, whose values are 16-bit unsigned integers representing Unicode characters. [...] The values of the integral types are integers in the following ranges: [...] For int, from −2147483648 to 2147483647, inclusive"

The Java `int` type and its range are formalized in Isabelle/HOL [Pau94] in the following.

First, we fix the bit length. This is the only constant in the whole theory that needs to be adapted if a different bit length (e.g. to represent the Java types `short`, `byte` etc.) is to be represented. Then, we fix the maximum and minimum values that Java integers can have.

constdefs
  $BitLength :: nat$
  $BitLength \equiv 32$
  $ld\text{-}BitLength :: nat$
  $ld\text{-}BitLength \equiv THE\ x.\ 2\hat{}x = BitLength$
  $MinInt\text{-}int :: int$
  $MinInt\text{-}int \equiv -\ (2\ \hat{}\ (BitLength - 1))$

MaxInt-int :: *int*
MaxInt-int ≡ *2*^(*BitLength* − *1*) − *1*

We introduce the new type of (normal behaviour) Java Integers by the range of the (mathematical) integers as follows:

typedef *JavaInt* = {*i*. *MinInt-int* ≤ *i* & *i* ≤ *MaxInt-int*}
by (*unfold MinInt-int-def MaxInt-int-def BitLength-def*, *auto*)

This construct is the Isabelle/HOL shortcut for a type definition which defines the new type *JavaInt* isomorphic to the set of integers between *MinInt-int* and *MaxInt-int*. The isomorphism is established through the automatically provided (total) functions *Abs-JavaInt* :: *int* ⇒ *JavaInt* and *Rep-JavaInt* :: *JavaInt* ⇒ *int* and the two axioms *Abs-JavaInt-inverse*: *y* ∈ *JavaInt* ⟹ *Rep-JavaInt* (*Abs-JavaInt y*) = *y*

and *Rep-JavaInt-inverse*: *Abs-JavaInt* (*Rep-JavaInt x*) = *x*.

Abs-JavaInt yields an arbitrary value if the argument is outside of the defining interval of *JavaInt*.

4.1 General Definitions

We define *MinInt* and *MaxInt* to be elements of the new type *JavaInt*:

constdefs
MinInt :: *JavaInt*
MinInt ≡ *Abs-JavaInt MinInt-int*
MaxInt :: *JavaInt*
MaxInt ≡ *Abs-JavaInt MaxInt-int*

In Java, calculations are only performed on values of the types int and long. Values of the three smaller integral types are widened first:

Java Language Specification [GJSB00], §4.2.2

> "If an integer operator other than a shift operator has at least one operand of type long, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type long. If the other operand is not long, it is first widened (§5.1.4) to type long by numeric promotion (§5.6). Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type int. If either operand is not an int, it is first widened to type int by numeric promotion. The built-in integer operators do not indicate overflow or underflow in any way."

This report describes the formalization of the Java type int, therefore conversions between the different numerical types are not in the focus of this work. The integer types byte and short can easily be added as all calculations are performed on the type int anyways, so the only operations that need to be implemented are the widening to int, and the cast operations from int to byte and short, respectively. The Java type long can be added equally

easily as our theory uses the bit length as a parameter, so one only need to change the definition of the bit length (see above) to gain a full theory for the Java type long, and again only the widening operations need to be added. Therefore, we only conentrate on the Java type int in the following.

Our model of Java int covers all side-effect-free operators. This excludes the operators ++ and −−, both in pre- and postfix notation. These operators return the value of the variable they are applied to while modifying the value stored in that variable independently from returning the value. We do not treat assignment of any kind either as it represents a side-effect as well. This also disallows combined operators like `a += b` etc. which are a shortcut for `a = a + b`. This is in line with usual specification languages, e.g. JML [LBR99], which also allows only side-effect-free operators in specifications. From a logical point of view, this makes sense as the specification is usually regarded as a set of predicates. In usual logics, predicates are side-effect-free. Thus, expressions with side-effects must be treated differently, either by special Hoare rules or by program transformation.

In our model, all operators are defined in Isabelle/HOL, and their properties as described in the JLS are proven, which ensures the validity of the definitions in our model. In the following, we quote the definitions from the JLS and present the Isabelle/HOL definitions and lemmas.

Our standard approach of defining the arithmetic operators on JavaInt is to convert the operands from JavaInt to Isabelle int, to apply the corresponding Isabelle int operation, and to convert the result back to JavaInt. The first conversion is performed by the representation function *Rep-JavaInt* (see above). The inverse conversion is performed by the function *Int-to-JavaInt*:

constdefs

$$
\begin{array}{l}
\textit{Int-to-JavaInt} :: \textit{int} \Rightarrow \textit{JavaInt} \\
\quad \textit{Int-to-JavaInt}\ (x{::}\textit{int}) \equiv \\
\qquad \textit{Abs-JavaInt}(\\
\qquad\quad ((x + (-\textit{MinInt-int}))\ \textit{mod}\ (2 * (-\textit{MinInt-int}))\ ) \\
+ \textit{MinInt-int}\ )
\end{array}
$$

This function first adds $-$ *MinInt-int* to the argument and then performs a modulo calculation by $2 * -$ *MinInt-int* which maps the value into the interval $[0\ ..\ 2 * -\textit{MinInt-int} - 1]$ (which is equivalent to only regarding the lowest 32 bits), and finally subtracts the value that was initially added. This definition is identical to the function *Abs-JavaInt* on arguments which are already in *JavaInt*. Larger or smaller values are mapped to *JavaInt* values, extending the domain to int.

This standard approach is not followed for operators that are explicitly defined on the bit representation of the arguments. Our approach differs from the approach used by Jacobs [Jac03] who exclusively uses bit representations for the integer representation as well as the operator definitions.

4.1.1 General Lemmas

These lemmas are about *MaxInt-int* and *MinInt-int* only.

lemma *MaxInt-MinInt-add*:
  MaxInt-int + *MinInt-int* = −*1*
  by (*simp add*: *MaxInt-int-def MinInt-int-def BitLength-def*)

lemma *Minustwo-Minus-two* :
(−*2*) ∗ *MinInt-int* = − (*2* ∗ *MinInt-int*)
  by *simp*

lemma *JavaInt-maxmin-compare*:
  MaxInt-int + *1* = − *MinInt-int*
  by (*unfold MaxInt-int-def MinInt-int-def BitLength-def*, *simp*)

lemma *JavaInt-maxmin-compare2*:
  MinInt-int = − (*MaxInt-int* + *1*)
  by (*unfold MaxInt-int-def MinInt-int-def BitLength-def*, *simp*)

lemma *JavaInt-maxmin-compare3*:
  MaxInt-int = − (*MinInt-int* + *1*)
  by (*unfold MaxInt-int-def MinInt-int-def BitLength-def*, *simp*)

lemma *null-leq-MaxInt* [*simp*] : *0* ≤ *MaxInt-int*
  by (*simp add*: *MaxInt-int-def BitLength-def*)

lemma *MinInt-leq0* [*simp*] : *MinInt-int* ≤ *0*
  by (*simp add*: *MinInt-int-def BitLength-def*)

lemma *MinInt-less0* [*simp*]: *MinInt-int* < *0*
  by (*unfold MinInt-int-def BitLength-def*, *simp*)

These lemmas relate *MaxInt-int* and *MinInt-int* to the new *JavaInt* type.

lemma *MaxInt-int-in-JavaInt* [*simp*] :
MaxInt-int : *JavaInt*
by (*simp add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)

lemma *MinInt-int-in-JavaInt* [*simp*] :
MinInt-int : *JavaInt*
by (*simp add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)

lemma *Rep-Abs-remove-general* :
  Rep-JavaInt (*Abs-JavaInt* (*x mod* − (*2* ∗ *MinInt-int*) +
  MinInt-int))
    = (*x mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
  apply (*rule Abs-JavaInt-inverse*)
  apply (*simp add*: *MinInt-int-def JavaInt-def MaxInt-int-def BitLength-def*)
  apply (*simp add*: *zle-add1-eq-le* [*symmetric*] *pos-mod-bound*)
done

lemma *Rep-Abs-remove* :
$(\bigwedge (x::int).$
$Rep\text{-}JavaInt\ (Abs\text{-}JavaInt\ ((x - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int))$
$= ((x - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int))$
apply (*rule Abs-JavaInt-inverse*)
apply (*simp add*: *MinInt-int-def JavaInt-def MaxInt-int-def BitLength-def*)
apply (*simp add*: *zle-add1-eq-le* [*symmetric*] *pos-mod-bound*)
done

lemma *MinInt-less-minus* [*simp*] : $MinInt\text{-}int < - Rep\text{-}JavaInt\ a$
by (*insert Rep-JavaInt*,
simp del: *zle-add1-eq-le add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def zle-add1-eq-le* [*symmetric*])

lemma *MinInt-leq-minus* [*simp*] : $MinInt\text{-}int \leq - Rep\text{-}JavaInt\ a$
apply (*insert MinInt-less-minus*)
apply (*rule zless-or-eq-imp-zle*)
apply *auto*
done

lemma *MinInt-leq* [*simp*] : $MinInt\text{-}int \leq Rep\text{-}JavaInt\ b$
by (*insert Rep-JavaInt*,
simp add: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)

lemma *MaxInt-geq* [*simp*] : $Rep\text{-}JavaInt\ b \leq MaxInt\text{-}int$
by (*insert Rep-JavaInt*,
simp add: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)

lemma *Int-to-JavaInt-charn*: $Rep\text{-}JavaInt(Int\text{-}to\text{-}JavaInt\ x) : JavaInt$
apply (*simp add*: *JavaInt-def*)
done

lemma *MinInt-leq-pos* [*simp*] : $MinInt\text{-}int \leq Rep\text{-}JavaInt\ a$
by (*simp add*: *Rep-JavaInt*)

lemma *MinInt-less-pos* :
$Rep\text{-}JavaInt\ a \neq MinInt\text{-}int \Longrightarrow MinInt\text{-}int < Rep\text{-}JavaInt\ a$
by (*simp add*: *int-less-le HOL.neq-commute*)

lemma *minus-leq-MaxInt-plus1* [*simp*] : − *Rep-JavaInt x* ≤ *MaxInt-int* + *1*
apply (*insert MinInt-leq*)
apply (*simp add*: *MaxInt-int-def BitLength-def JavaInt-def MinInt-int-def*)
done

lemma *minus-leq-MaxInt* : **assumes** *1*: *Rep-JavaInt x* ≠ *MinInt-int*
shows − *Rep-JavaInt x* ≤ *MaxInt-int*
apply (*simp add*: *zminus-zle* [*of Rep-JavaInt x MaxInt-int*])
apply (*simp add*: *JavaInt-maxmin-compare3*)
apply (*simp add*: *add1-zle-eq*)
apply (*cut-tac b* = *x* **in** *MinInt-leq*)
apply (*insert 1* [*symmetric*])
apply (*drule zle-imp-zless-or-eq*)
apply *simp*
done

lemma *minus-in-JavaInt2*:
assumes *1*: *x* : *JavaInt*
assumes *2*: *x* ≠ *MinInt-int*
shows − *x* : *JavaInt*

apply (*simp add*: *JavaInt-def*)

apply (*simp add*: *zminus-zle* [*of x MaxInt-int*])
apply (*simp add*: *JavaInt-maxmin-compare3*)
apply (*simp add*: *add1-zle-eq*)

apply (*simp add*: *zle-zminus* [*of MinInt-int x*])
apply (*simp add*: *JavaInt-maxmin-compare*[*symmetric*])

apply (*insert 1*)
apply (*simp add*: *JavaInt-def*)
apply (*insert 2*)
apply *auto*
done

lemma *minus-in-JavaInt*:
assumes *1*: *Rep-JavaInt x* ≠ *MinInt-int*
shows − *Rep-JavaInt x* : *JavaInt*
by (*insert 1*, *simp add*: *minus-in-JavaInt2 Rep-JavaInt*)

lemma *pos-minus-in-JavaInt* [*simp*] :
assumes *1*: *0* < *Rep-JavaInt b*
shows − *Rep-JavaInt b* : *JavaInt*

```
    apply (rule minus-in-JavaInt)
    apply (insert MinInt-leq0 1)
    apply (drule order-le-less-trans [of MinInt-int 0 Rep-JavaInt b],
      simp)
    apply simp
    done

lemma NotMinInt1: assumes 1: x ≠ MinInt
  shows abs (Rep-JavaInt x) : JavaInt
  apply (insert 1)
    apply (simp add: zabs-def)
    apply auto
    apply (simp-all add: Rep-JavaInt)
    apply (simp add: JavaInt-def)
    apply (simp add: zminus-zle)
    apply (simp add: MinInt-def)
    apply (drule Rep-JavaInt-inverse [of x, symmetric,THEN subst])
    apply rotate-tac
    apply (simp add: Abs-JavaInt-inject MinInt-int-in-JavaInt Rep-JavaInt)
    apply (simp add: zminus-zle [of MaxInt-int Rep-JavaInt x])
    apply (rule minus-leq-MaxInt, simp)
    done


lemma Not-MinInt-minus-leq-MaxInt:
          assumes 1: Rep-JavaInt x ≠ MinInt-int
  shows − Rep-JavaInt x ≤ MaxInt-int
  apply (insert 1)
  apply (subst zle-add1-eq-le [of − Rep-JavaInt x MaxInt-int, symmetric])
  apply (simp only: JavaInt-maxmin-compare)
  apply (simp only: zminus-zless-zminus)
  apply (cut-tac b = x in MinInt-leq)
  apply rotate-tac
  apply (drule zle-imp-zless-or-eq, auto)
  done

lemma Abs-Rep-Mod-identity2:
  Abs-JavaInt
      ((Rep-JavaInt a − MinInt-int) mod − (2 ∗ MinInt-int) + MinInt-int) = a
  apply (rule mod-pos-pos-trivial [symmetric, THEN subst])
  apply (subst zle-zdiff-eq, simp)
  apply (simp add: zdiff-zless-eq)
  apply (simp add: zless-zminus)
  apply (simp add: Rep-JavaInt-inverse)
  done

lemma Abs-Rep-Mod-identity-abstract:
   Int-to-JavaInt (Rep-JavaInt x) = x
  by (unfold Int-to-JavaInt-def, simp add: Abs-Rep-Mod-identity2)
```

lemma *Int-to-JavaInt-ident* :
  assumes *1*: x : *JavaInt*
  shows (*Int-to-JavaInt* x) = *Abs-JavaInt* (x::*int*)
proof −
  have *Abs-JavaInt*
        (((*x* − *MinInt-int*) *mod* (− (*2* ∗ *MinInt-int*))) + *MinInt-int*)
      = *Abs-JavaInt* ((*x* − *MinInt-int*) + *MinInt-int*)
    apply (*insert 1*)
  apply (*simp add*: *JavaInt-def*)
  apply *auto*
  apply (*subst mod-pos-pos-trivial*)
  apply *auto*
  apply (*simp add*: *MinInt-int-def MaxInt-int-def*)
  done
  also have ... = *Abs-JavaInt x* **by** *auto*
  also from *calculation* **show** *?thesis*
  by (*simp add*: *Int-to-JavaInt-def JavaInt-def*)
qed

lemma *not-MinInt-not-MinInt-int*:
shows ($b \neq$ *MinInt*) = ((*Rep-JavaInt b*) ≠ *MinInt-int*)
  by (*simp add*: *Rep-JavaInt-inject*[*symmetric*] *MinInt-def Abs-JavaInt-inverse*)

The following lemmas only talk about elements of *JavaInt*.

lemma *eq-commute*:
  ((a::*JavaInt*) = b) = (b = a)
  by (*simp add*: *Rep-JavaInt-inject*[*symmetric*]
    HOL.eq-commute[*of Rep-JavaInt a Rep-JavaInt b*])

lemma *neq-commute*:
  ((a::*JavaInt*) ≠ b) = (b ≠ a)
  by (*simp add*: *Rep-JavaInt-inject*[*symmetric*]
    HOL.neq-commute[*of Rep-JavaInt a Rep-JavaInt b*])

4.2 Axclass zero

lemma *JavaInt-in-zero*:
OFCLASS(*JavaInt*, *zero-class*)
apply (*rule zero.intro*)
..

instance *JavaInt* :: *zero* **..**

defs (**overloaded**)
  zero-def : *0* ≡ *Abs-JavaInt 0*

4.2.1 zero Lemmas

lemma *zero-in-JavaInt* [*simp*] :
0 : *JavaInt*
by (*unfold JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*, *auto*)

lemma *Rep-Abs-zero* [*simp*] : *Rep-JavaInt* (*Abs-JavaInt 0*) = *0*
by (*simp add*: *Abs-JavaInt-inverse zero-in-JavaInt*)

lemma *Int-to-JavaInt-zero-zero* [*simp*] : *Int-to-JavaInt 0* = *0*
by (*simp add*: *Int-to-JavaInt-ident zero-def*)

4.3 Axclass one

lemma *JavaInt-in-one*:
OFCLASS(*JavaInt*, *one-class*)
apply (*rule one.intro*)
..

instance *JavaInt* :: *one* **..**

defs (**overloaded**)
one-def : *1* ≡ *Abs-JavaInt 1*

4.3.1 one Lemmas

lemma *one-in-JavaInt* [*simp*] :
1 : *JavaInt*
by (*simp add*: *JavaInt-def one-def MinInt-int-def MaxInt-int-def BitLength-def*)

4.4 Axclass number

Required for bit string representation.

instance *JavaInt* :: *number* **..**

defs (**overloaded**)
number-of-def : (*number-of*:: *bin* ⇒ *JavaInt*) *b*
≡ *Int-to-JavaInt* ((*number-of*::*bin* ⇒ *int*) *b*)

The inverse operation *bin-of* is defined in Sec. 17 together with the other bitstring operations.

end

theory *JavaIntegersAdd* = *JavaIntegersDef* :

5 Addition and Subtraction

This section formalizes the unary and binary + and − operators.

The JLS describes the binary + and − operators as follows:

> Java Language Specification [GJSB00], §15.18.2
>
> "The binary + operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The binary - operator performs subtraction, producing the difference of two numeric operands.(1) [...]
> Addition is a commutative operation if the operand expressions have no side effects. Integer addition is associative when the operands are all of the same type(2) [...]
> If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format.(3)
> If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.(4)
> For both integer and floating-point subtraction, it is always the case that a−b produces the same result as a+(−b).(5)"

These two operators are defined in the standard way described above. Their definitions are given below.

5.1 Axclass plus

instance *JavaInt* :: *plus* **..**

This is the definition of the binary + operator that relates to (1) in the JLS, §15.18.2:

defs (**overloaded**)
  add-def : (x::*JavaInt*) + y ≡
    Int-to-JavaInt (*Rep-JavaInt* x + *Rep-JavaInt* y)

This definition of the addition already captures property (3) in the JLS, §15.18.2.

The unary plus operator on `int` is equivalent to the identity function. This is not very challenging, thus we do not elaborate on this operator.

constdefs
  uplus :: *JavaInt* ⇒ *JavaInt*

  uplus (x::*JavaInt*) ≡ x

5.1.1 plus Lemmas

lemma *wraparound-top* [*simp*] : *MaxInt* + *1* = *MinInt*
proof −
  have *MaxInt* + *1* =

```
    Int-to-JavaInt( Rep-JavaInt MaxInt + Rep-JavaInt 1)
    by (simp add: add-def)
  also have ... =
    Int-to-JavaInt(Rep-JavaInt(Abs-JavaInt MaxInt-int) +
                   Rep-JavaInt(Abs-JavaInt 1))
    by (simp add: MaxInt-def one-def)
  also have ... =
    Int-to-JavaInt(MaxInt-int + 1)
    by (simp add: Abs-JavaInt-inverse)
  also have ... = Abs-JavaInt
     ((MaxInt-int + 1 − MinInt-int) mod − (2 ∗ MinInt-int) + MinInt-int)
    by (simp add: Int-to-JavaInt-def)
  also have ... = Abs-JavaInt MinInt-int
    by (simp add: MaxInt-int-def MinInt-int-def BitLength-def)
  also have ... = MinInt
    by (simp add: MinInt-def [symmetric])
  also from calculation show ?thesis .
qed
```

```
lemma JavaInt-add-left-neutral [simp] :
  (0::JavaInt) + x = x
  apply (unfold add-def zero-def)
  apply (subst Abs-JavaInt-inverse)
  apply (simp-all add: Abs-Rep-Mod-identity-abstract)
done
```

This lemma captures the first property described in (2) in the JLS, §15.18.2:

```
lemma JavaInt-add-commute:
  x + y = y + (x::JavaInt)
  by (simp add: add-def zadd-commute)

lemma Int-to-JavaInt-homom [simp] :
Int-to-JavaInt (Rep-JavaInt ((Int-to-JavaInt a) + (Int-to-JavaInt b))) =
 Int-to-JavaInt (a + b)
proof −
  have ∗ : Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) + Rep-JavaInt (Int-to-JavaInt
b)) =
    Int-to-JavaInt (a + b)
  apply (simp add: Int-to-JavaInt-def)
  apply (simp add: Rep-Abs-remove )

  apply (simp add: zadd-assoc [symmetric])
  apply (simp add: zmod-zadd-right-eq [symmetric])
  apply (simp add: zmod-zadd-left-eq [symmetric])
  apply (simp add: zdiff-zadd-eq )
  done
```

```
  show ?thesis
    apply (simp add: add-def)
    apply (simp add: Rep-JavaInt Int-to-JavaInt-ident
      [of (Rep-JavaInt
     (Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) + Rep-JavaInt (Int-to-JavaInt
b))))])
    apply (simp add: Rep-JavaInt-inverse * )
  done
qed
```

This lemma captures the second property described in (2) in the JLS, §15.18.2:

```
lemma JavaInt-add-assoc:
  x + y + z = x+(y+z::JavaInt)
proof -
  have ** : - (2 * MinInt-int) = MaxInt-int + 1 - MinInt-int
    by (simp add: MinInt-int-def MaxInt-int-def BitLength-def)

   have [intro]: ⋀a b. Int-to-JavaInt (a + (Rep-JavaInt(Int-to-JavaInt b))) =
Int-to-JavaInt (a+b)
  proof -
    fix a b
   have Int-to-JavaInt b = Abs-JavaInt ((b - MinInt-int) mod - (2 * MinInt-int)
+ MinInt-int)
      (is - = Abs-JavaInt ?x)
      by (simp add: Int-to-JavaInt-def)
    moreover
    have ?x : JavaInt
      apply (simp add: pos-mod-sign JavaInt-def)
     apply (simp del: zle-add1-eq-le Minustwo-Minus-two add: pos-mod-sign zle-add1-eq-le
[symmetric] **
                zless-zdiff-eq [symmetric] pos-mod-bound JavaInt-maxmin-compare)
      done
    ultimately
    have Rep-JavaInt(Int-to-JavaInt b) = ?x
      by (simp add: Abs-JavaInt-inverse)
    thus ?thesis a b
      apply (simp add: Int-to-JavaInt-def)
     apply (simp only: zmod-zadd-right-eq [symmetric] int-diff-minus-eq zadd-zdiff-eq
zadd-assoc)
      done
  qed
  moreover
  have ⋀a b. Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) + b) = Int-to-JavaInt
(a+b)
  proof -
    fix a b
    have Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) + b) =
            Int-to-JavaInt (b + Rep-JavaInt (Int-to-JavaInt a)) by (simp add:
```

zadd-commute)
also
have ... = *Int-to-JavaInt* (*b*+*a*) **..**
also
have ... = *Int-to-JavaInt* (*a*+*b*) **by** (*simp add*: *zadd-commute*)
also from *calculation*
show *?thesis a b* **.**
qed
ultimately
show *?thesis* **by** (*unfold add-def*, *simp add*: *zadd-assoc*)
qed

The remaining properties are given in Sec. 12.5 because they require comparison operators which have not been formalized yet.

5.2 Axclass minus

instance *JavaInt* :: *minus* **..**

This is the definition of the binary − operator that relates to (1) in the JLS, §15.18.2:

defs (**overloaded**)
diff-def : (*x*::*JavaInt*) − *y* ≡
Int-to-JavaInt (*Rep-JavaInt x* − *Rep-JavaInt y*)

In the JLS, the unary minus operator is defined in relation to the binary minus operator described above.

Java Language Specification [GJSB00], §15.15.4

"At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand. For integer values, negation is the same as subtraction from zero.(1)
[...] negation of the maximum negative int or long results in that same maximum negative number.(2)
[...] For all integer values x, −x equals (∼x)+1.(3)"

The unary minus operator is formalized as

defs (**overloaded**)
uminus-def : − (*x*::*JavaInt*) ≡ *Int-to-JavaInt* (− *Rep-JavaInt x*)

Note that the unary minus operator has two fixed points: 0 and MinInt. This leads to some unexpected results, e.g. `Math.abs(MinInt)` = MinInt, a negative number (see Sec. 12.4 for details). Also, many of the lemmas presented in this report do not hold for MinInt and therefore exclude that value in their assumptions.

The bitwise complement operator is defined by unary and binary minus:

Java Language Specification [GJSB00], §15.15.5

> "At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, ∼x equals (−x)−1."

This is formalized in Isabelle/HOL as follows:

constdefs
  JavaInt-bitcomplement :: *JavaInt* ⇒ *JavaInt*
    JavaInt-bitcomplement (*x*::*JavaInt*) ≡ (−*x*) − (*1*::*JavaInt*)

For the Java types byte, short or char, the argument is promoted to int first.

We use the notations ∼ and *JavaInt-bitcomplement* interchangeably.

5.2.1 minus Lemmas

This is the first property described for the unary minus operator in the JLS:

lemma *uminus-property*: *0* − *x* = − (*x*::*JavaInt*)
  by (*simp add*: *uminus-def diff-def zero-def*
    Abs-JavaInt-inverse zero-in-JavaInt)

lemma *wraparound-bottom*:
MinInt − *1* = *MaxInt*
apply (*simp add*: *diff-def MinInt-def MaxInt-def Int-to-JavaInt-def one-def*)
apply (*simp add*: *Abs-JavaInt-inverse*)
apply (*simp add*: *MinInt-int-def MaxInt-int-def BitLength-def*)
done

lemma *JavaInt-add-minus-inverse*:
(−(*a*::*JavaInt*)) + *a* = *0*
  apply (*unfold add-def uminus-def*)
  apply (*unfold Int-to-JavaInt-def zero-def*)
  apply (*simp add*: *Rep-Abs-remove*)
  apply (*simp add*: *zmod-zadd-left-eq* [*symmetric*])
  apply (*simp add*: *MinInt-int-def BitLength-def*)
done

lemma *uminus-one-is-minusone* [*simp*]: − (*1*::*JavaInt*) = −*1*
 apply (*simp-all only* : *JavaIntegersDef.number-of-def JavaIntegersAdd.uminus-def*
                JavaIntegersDef.one-def)
  apply (*subst Abs-JavaInt-inverse*)
  apply (*simp-all*)
  done

This is the second property described for the unary minus operator in the JLS:

lemma *uminus-MinInt*:
  − *MinInt* = *MinInt*

```
  proof −
    have
        Abs-JavaInt ((− MinInt-int − MinInt-int) mod − (2 ∗ MinInt-int) +
MinInt-int) =
      Abs-JavaInt ((− MinInt-int + (− MinInt-int)) mod − (2 ∗ MinInt-int) +
MinInt-int)
      (is - = Abs-JavaInt (( − ?mi + (− ?mi)) mod −(2∗?mi) + ?mi))
      by auto
    also have
      ... = Abs-JavaInt ((− (?mi + ?mi)) mod −(2∗?mi) + ?mi)
      by (simp only: zminus-zadd-distrib )
    also have
      ... = Abs-JavaInt ((− (2 ∗ ?mi)) mod − (2 ∗ ?mi) + ?mi)
      by auto
    also have
        ... = Abs-JavaInt ?mi
        by (simp only: zmod-self, auto)
    also from calculation
    show ?thesis
    apply (simp add: MinInt-def uminus-def Int-to-JavaInt-def Abs-JavaInt-inverse)
    done
  qed


lemma uminus-uminus-ident [simp] :
  − (− (x::JavaInt)) = x
  apply (cases x = MinInt)
  apply (simp add: uminus-MinInt)
  apply (simp add: uminus-def)
  apply (simp add: Rep-JavaInt-inject[symmetric] MinInt-def
    Abs-JavaInt-inverse)
  apply (simp add: Int-to-JavaInt-ident minus-in-JavaInt)
  apply (simp add: Abs-JavaInt-inverse minus-in-JavaInt)
  apply (simp add: Abs-Rep-Mod-identity-abstract)
  done

lemma minus-zero [simp] :
  − 0 = (0::JavaInt)
  by (simp add: zero-def uminus-def
    Rep-JavaInt-inject[symmetric] Abs-JavaInt-inverse)
```

This lemma captures the property described in (5) in the JLS, §15.18.2:

```
lemma diff-uminus: a − b = a + (−b::JavaInt)
proof −
  have Abs-JavaInt
    ((Rep-JavaInt a − Rep-JavaInt b − MinInt-int) mod − (2 ∗ MinInt-int) +
     MinInt-int) = Abs-JavaInt
    ((Rep-JavaInt a + (−Rep-JavaInt b) − MinInt-int) mod − (2 ∗ MinInt-int)
+
```

```
    MinInt-int)
    (is - = Abs-JavaInt ((?ra + ?rb − ?m) mod ?mm + ?m))
  by auto
  also have ... =
          Abs-JavaInt ((?ra + (?rb − ?m)) mod ?mm + ?m)
  by (simp only: zadd-zdiff-eq [symmetric])
  also have ... =
          Abs-JavaInt ((?ra + (?rb − ?m) mod ?mm) mod ?mm + ?m)
  by (simp only: zmod-zadd-right-eq [symmetric])
  also have ... =
          Abs-JavaInt ((?ra + ((?rb − ?m) mod ?mm + ?m − ?m)) mod ?mm +
?m)
  by auto
  also have ... =
          Abs-JavaInt ((?ra + ((?rb − ?m) mod ?mm + ?m) − ?m) mod ?mm +
?m)
  by auto
  also from calculation
  show ?thesis
    apply (simp add: uminus-def diff-def add-def Int-to-JavaInt-def)
    apply (simp only: Rep-Abs-remove)
    apply auto
  done
qed

lemma Int-to-JavaInt-neg-MinInt-int [simp] : Int-to-JavaInt (− MinInt-int) =
MinInt
  apply (simp add: JavaInt-maxmin-compare[symmetric])
  apply (simp add: Int-to-JavaInt-def)
  apply (simp add: MaxInt-int-def MinInt-def MinInt-int-def BitLength-def)
  done

lemma minusone-in-JavaInt [simp] : −1 : JavaInt
    by (simp add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)

lemma Abs-minusone [simp]: Abs-JavaInt −1 = −1
    by (simp add: number-of-def Int-to-JavaInt-ident)

lemma Rep-Int-minusone [simp] : Rep-JavaInt (Int-to-JavaInt −1) = −1
  apply (subst Int-to-JavaInt-ident)
  defer
  apply (subst Abs-JavaInt-inverse)
  apply (tactic distinct-subgoals-tac)
  apply (simp add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  apply simp
  done


lemma uminus-homom :
```

assumes *1*: $a \neq$ *MinInt*
shows *Rep-JavaInt* $(-\ a) = -$ *Rep-JavaInt* (a)
apply (*simp add*: *uminus-def*)
apply (*insert 1*)
apply (*simp add*: *Rep-JavaInt-inject*[*symmetric*] *MinInt-def Abs-JavaInt-inverse*)
apply (*simp add*: *Int-to-JavaInt-ident minus-in-JavaInt Abs-JavaInt-inverse*)
done

lemma *uminus-shift*:
$(-\ x = (y{::}JavaInt)) = (x = -\ y)$
by *auto*

lemma *uminus-eq-zero-eq-zero* [*simp*]: $(-\ xa = (0{::}JavaInt)) = (xa = 0)$
by (*simp add*: *uminus-shift*)

lemma *neg-not-MinInt*:
$((-\ d) \neq MinInt) = (d \neq MinInt)$
by (*auto simp*: *uminus-MinInt uminus-shift*)

This is the third property described for the unary minus operator in the JLS:

lemma *uminus-bitcomplement*: (*JavaInt-bitcomplement x*) + *1* = − *x*
proof −
have *Int-to-JavaInt* (− *Rep-JavaInt x*) − *1* + *1* =
Int-to-JavaInt (− *Rep-JavaInt x*) + (− *1*) + *1*
by (*simp add*: *diff-uminus*)
also have ... = *Int-to-JavaInt* (− *Rep-JavaInt x*) + (− *1* + *1*)
by (*simp only*: *JavaInt-add-assoc*)
also have ... = *Int-to-JavaInt* (− *Rep-JavaInt x*)
apply (*simp only*: *JavaInt-add-minus-inverse*)
apply (*simp only*: *JavaInt-add-commute*
JavaInt-add-left-neutral)
done
also from *calculation* **show** *?thesis*
by (*simp add*: *JavaInt-bitcomplement-def uminus-def*)
qed

5.3 Axclass plus_ac0

instance *JavaInt* :: *plus-ac0*
apply *intro-classes*
apply (*rule JavaInt-add-commute*)
apply (*rule JavaInt-add-assoc*)
apply (*rule JavaInt-add-left-neutral*)
done

end

theory *JavaIntegersTimes* = *JavaIntegersAdd* :

6 Multiplication

6.1 Axclass times

instance *JavaInt* :: *times* **..**

The multiplication operator is described and formalized as follows:

> Java Language Specification [GJSB00], §15.17.1
>
> "The binary * operator performs multiplication, producing the product of its operands. Multiplication is a commutative operation if the operand expressions have no side effects. [...] integer multiplication is associative when the operands are all of the same type"

defs (**overloaded**)
times-def : $x * y \equiv$
Int-to-JavaInt (*Rep-JavaInt* x $*$ *Rep-JavaInt* y)

> Java Language Specification [GJSB00], §15.17.1
>
> "If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values."

This is again implicitly fulfilled by our standard modelling.

6.1.1 times Lemmas

lemma *JavaInt-times-one-ident*:
1 $*$ (a::*JavaInt*) = a
apply (*unfold one-def times-def*, *simp add*: *Int-to-JavaInt-def one-def*)
apply (*subst Abs-JavaInt-inverse*)
apply (*simp add*: *one-in-JavaInt*)
apply (*simp add*: *Abs-Rep-Mod-identity2*)
done

lemma *eq-times-eq*:
assumes *1*: (x::*JavaInt*) = y
shows $x * z = y * z$
apply (*simp add*: *times-def*)
apply (*insert 1*)

apply (*simp only*: *Rep-JavaInt-inject*[*symmetric*, *of x y*])
done

lemma *eq-times-eq2*:
assumes *1*: $x = (y::JavaInt)$
shows $z * x = z * y$
apply (*insert 1*)
apply (*simp add*: *times-def*)
done

lemma *uminus-times-minusone*:
$-1 * x = -(x::JavaInt)$
by (*simp add*: *times-def uminus-def number-of-def*)

The commutativity of the multiplication operator is proven by the lemma

lemma *JavaInt-times-commute*:
$(x::JavaInt) * y = y * x$
by (*simp add*: *times-def zmult-commute*)

lemma *MinInt-times-minusone* :
$MinInt * -1 = MinInt$
apply (*subst JavaInt-times-commute*)
apply (*simp add*: *uminus-times-minusone uminus-MinInt*)
done

The associativity of the multiplication operator is proven by the lemma

lemma *JavaInt-times-assoc*:
$x * y * z = x*(y*z::JavaInt)$

proof −
{
fix *a b*
have
Abs-JavaInt
$((a * ((b - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int) - MinInt\text{-}int)$
mod
$- (2 * MinInt\text{-}int) +$
$MinInt\text{-}int) =$
$Abs\text{-}JavaInt\ ((a * b - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int)$
proof −
have *Abs-JavaInt*
$((a * ((b - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int) - MinInt\text{-}int)$
mod
$- (2 * MinInt\text{-}int) +$
$MinInt\text{-}int) = Abs\text{-}JavaInt$
$((a * ((b - MinInt\text{-}int)\ mod - (2 * MinInt\text{-}int) + MinInt\text{-}int) + (-$
$MinInt\text{-}int))\ mod$
$- (2 * MinInt\text{-}int) +$

MinInt-int) **by** *auto*
also
have ... =
Abs-JavaInt
(((*a* ∗ ((*b* − *MinInt-int*) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)) *mod* −(*2*∗*MinInt-int*)
+ (−*MinInt-int*)) *mod*
− (*2* ∗ *MinInt-int*) +
MinInt-int)
by (*simp only*: *zmod-zadd-left-eq* [*symmetric*])
also
have ... = *Abs-JavaInt*
((
(*a* ∗ ((*b* − *MinInt-int*) *mod* − (*2*∗*MinInt-int*)) + *a* ∗ *MinInt-int*)
mod −(*2*∗*MinInt-int*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp only*: *zadd-zmult-distrib2*)
also
have ... = *Abs-JavaInt*
((
(*a* ∗ ((*b* − *MinInt-int*) *mod* −(*2*∗*MinInt-int*)) *mod* −(*2*∗*MinInt-int*) + *a* ∗ *MinInt-int*)
mod −(*2*∗*MinInt-int*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp add*: *zmod-zadd-left-eq* [*symmetric*])
also
have ... = *Abs-JavaInt*
((
((*a* ∗(*b* − *MinInt-int*)) *mod* −(*2*∗*MinInt-int*) + *a* ∗ *MinInt-int*)
mod −(*2*∗*MinInt-int*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp only*: *zmod-zmult1-eq* [*symmetric*])
also
have ... = *Abs-JavaInt*
((
((*a* ∗(*b* − *MinInt-int*)) + *a* ∗ *MinInt-int*)
mod −(*2*∗*MinInt-int*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp only*: *zmod-zadd-left-eq* [*symmetric*])
also
have ... = *Abs-JavaInt*
((
(*a*∗*b*)
mod −(*2*∗*MinInt-int*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp add*: *zdiff-zmult-distrib2*)
also
have ... = *Abs-JavaInt*
((
(*a*∗*b*) + (−*MinInt-int*)) *mod* − (*2* ∗ *MinInt-int*) + *MinInt-int*)
by (*simp only*: *zmod-zadd-left-eq* [*symmetric*])
also
from *calculation*
show *?thesis* **by** *auto*

```
  qed
} note Core-times-assoc = this

  have ** : − (2 ∗ MinInt-int) = MaxInt-int + 1 − MinInt-int
    by (simp add: MinInt-int-def MaxInt-int-def BitLength-def)

   have [intro]: ⋀a b. Int-to-JavaInt (a ∗ (Rep-JavaInt(Int-to-JavaInt b))) =
Int-to-JavaInt (a∗b)
  proof −
    fix a b
   have Int-to-JavaInt b = Abs-JavaInt ((b − MinInt-int) mod − (2 ∗ MinInt-int)
+ MinInt-int)
      (is - = Abs-JavaInt ?x)
      by (simp add: Int-to-JavaInt-def)
    moreover
    have ?x : JavaInt
      apply (simp add: pos-mod-sign JavaInt-def)
     apply (simp del: zle-add1-eq-le Minustwo-Minus-two add: pos-mod-sign zle-add1-eq-le
[symmetric] **
        zless-zdiff-eq [symmetric] pos-mod-bound JavaInt-maxmin-compare)
      done
    ultimately
    have Rep-JavaInt(Int-to-JavaInt b) = ?x
      by (simp add: Abs-JavaInt-inverse)
    thus ?thesis a b
      apply (simp add: Int-to-JavaInt-def)
      apply (simp only: Core-times-assoc )
      done
  qed

  moreover
  have ⋀a b. Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) ∗ b) = Int-to-JavaInt
(a∗b)
  proof −
    fix a b
    have Int-to-JavaInt (Rep-JavaInt (Int-to-JavaInt a) ∗ b) =
    Int-to-JavaInt (b ∗ Rep-JavaInt (Int-to-JavaInt a)) by (simp add: zmult-commute)
    also
    have ... = Int-to-JavaInt (b∗a) ..
    also
    have ... = Int-to-JavaInt (a∗b) by (simp add: zmult-commute)
    also from calculation
    show ?thesis a b .
  qed
  ultimately
  show ?thesis by (unfold times-def, simp add: zmult-assoc)
qed
```

The following lemma does not belong to this section thematically, but its

proof is performed very elegantly when multiplication is used, therefore we postponed it to here:

```
lemma uminus-zero-zero:
  assumes 1: − x = 0
  shows x = (0::JavaInt)
  apply (insert 1)
  apply (drule eq-times-eq [of − x 0 −1])
  apply (simp add: JavaInt-times-commute [of - −1])
  apply (simp add: uminus-times-minusone)
  done


lemma JavaInt-add-times-left-distrib:
  (a + b) ∗ c = a ∗ c + b ∗ (c::JavaInt)
proof −
  have Abs-JavaInt ((
    ((Rep-JavaInt a + Rep-JavaInt b − MinInt-int) mod − (2 ∗ MinInt-int) +
MinInt-int)
    ∗ Rep-JavaInt c − MinInt-int) mod − (2 ∗ MinInt-int) + MinInt-int)
    =
    Abs-JavaInt ((
    ((Rep-JavaInt a + Rep-JavaInt b + (− MinInt-int)) mod − (2 ∗ MinInt-int)
+ MinInt-int)
    ∗ Rep-JavaInt c + (− MinInt-int)) mod − (2 ∗ MinInt-int) + MinInt-int)
    (is Abs-JavaInt ((((  ?ab − ?mi  ) mod ?m + ?mi) ∗ ?c − ?mi  ) mod ?m +
?mi)
      = Abs-JavaInt ((?x + ?mmi) mod ?m + ?mi))
    by auto
  also have ... =
    Abs-JavaInt ((  ?x       mod ?m + ?mmi ) mod ?m + ?mi)
    (is - = Abs-JavaInt (((?y ∗ ?c) mod ?m + ?mmi ) mod ?m + ?mi))
    by (simp only: zmod-zadd-left-eq [symmetric])
  also have ... =
    Abs-JavaInt (((  ?y                  mod ?m ∗ ?c) mod ?m + ?mmi ) mod ?m +
?mi)
    (is - = Abs-JavaInt ((((?abm mod ?m + ?mi) mod ?m ∗ ?c) mod ?m + ?mmi
) mod ?m + ?mi))
    by (simp only: zmod-zmult1-eq′ [symmetric])
  also have ... =
    Abs-JavaInt ((((?abm       + ?mi) mod ?m ∗ ?c) mod ?m + ?mmi ) mod ?m
+ ?mi)
    (is - = Abs-JavaInt ((((?ab + ?mmi + ?mi) mod ?m ∗ ?c) mod ?m + ?mmi
) mod ?m + ?mi))
    by (simp only: zmod-zadd-left-eq [symmetric])
  also have ... =
    Abs-JavaInt (((  ?ab      ∗ ?c) mod ?m + ?mmi ) mod ?m + ?mi)
    (is - = Abs-JavaInt ((((?a + ?b) ∗ ?c) mod ?m + ?mmi ) mod ?m + ?mi))
    by (simp only: zmod-zmult1-eq′ [symmetric], auto)
  also have ... =
```

Abs-JavaInt (((*?a* ∗ *?c* + *?b* ∗ *?c*) *mod ?m* + *?mmi*) *mod ?m* + *?mi*)
by (*simp add*: *zadd-zmult-distrib*)
also have ... =
Abs-JavaInt (((*?a* ∗ *?c* + *?b* ∗ *?c*) + *?mmi*) *mod ?m* + *?mi*)
by (*simp only*: *zmod-zadd-left-eq* [*symmetric*])
also have ... =
Abs-JavaInt ((*?mi* + *?mmi* + *?a* ∗ *?c* + *?b* ∗ *?c* + *?mmi*) *mod ?m* + *?mi*)
by *auto*
also have ... =
Abs-JavaInt ((*?mi* + *?a* ∗ *?c* + *?mmi* + *?b* ∗ *?c* + *?mmi*) *mod ?m* + *?mi*)
by (*simp add*: *zadd-commute*)
also have ... =
Abs-JavaInt ((*?mi* + ((*?a* ∗ *?c* + *?mmi*) + (*?b* ∗ *?c* + *?mmi*)) *mod ?m*) *mod ?m* + *?mi*)
(**is** - = *Abs-JavaInt* ((*?mi* + (*?acm* + *?bcm*) *mod ?m*) *mod ?m* + *?mi*))
by (*simp only*: *zmod-zadd-right-eq* [*symmetric*] *zadd-assoc*)
also have ... =
Abs-JavaInt ((*?mi* + (*?acm mod ?m* + *?bcm mod ?m*) *mod ?m*) *mod ?m* + *?mi*)
(**is** - = *Abs-JavaInt* ((*?mi* + *?z* *mod ?m*) *mod ?m* + *?mi*))
by (*simp only*: *zmod-zadd1-eq* [*symmetric*])
also have ... = *Abs-JavaInt* ((*?mi* + *?z*) *mod ?m* + *?mi*)
by (*subst zmod-zadd-right-eq* [*symmetric*], *auto*)

also from *calculation*
show *?thesis*
by (*simp add*: *add-def times-def Int-to-JavaInt-def Rep-Abs-remove*)
qed

end

7 The algebraic hierarchy of rings as axiomatic classes

theory *Ring* = *Main*
files (*order.ML*):

8 Constants

Most constants are already declared by HOL.

consts
assoc :: [*'a*::*times*, *'a*] => *bool* (**infixl** *50*)
irred :: *'a*::{*zero*, *one*, *times*} => *bool*
prime :: *'a*::{*zero*, *one*, *times*} => *bool*

9 Axioms

9.1 Ring axioms

axclass *ring* < *zero*, *one*, *plus*, *minus*, *times*, *inverse*, *power*

a-assoc: $(a + b) + c = a + (b + c)$
l-zero: $0 + a = a$
l-neg: $(-a) + a = 0$
a-comm: $a + b = b + a$

m-assoc: $(a * b) * c = a * (b * c)$
l-one: $1 * a = a$

l-distr: $(a + b) * c = a * c + b * c$

m-comm: $a * b = b * a$

— Definition of derived operations

minus-def: $a - b = a + (-b)$
inverse-def: *inverse a* = (*if a dvd 1 then THE x. a*x = 1 else 0*)
divide-def: *a / b = a * inverse b*
power-def: *a ^ n = nat-rec 1 (%u b. b * a) n*

defs
assoc-def: *a assoc b == a dvd b & b dvd a*
irred-def: *irred a == a ~= 0 & ~ a dvd 1*
& (ALL d. d dvd a --> d dvd 1 | a dvd d)
prime-def: *prime p == p ~= 0 & ~ p dvd 1*
*& (ALL a b. p dvd (a*b) --> p dvd a | p dvd b)*

9.2 Integral domains

axclass
domain < *ring*

one-not-zero: *1 ~= 0*
integral: *a * b = 0 ==> a = 0 | b = 0*

9.3 Factorial domains

axclass
factorial < *domain*

factorial-divisor: *True*
factorial-prime: *irred a ==> prime a*

9.4 Euclidean domains

9.5 Fields

axclass
field < *ring*

field-one-not-zero: *1* ~= *0*

field-ax: *a* ~= *0* ==> *a dvd 1*

10 Basic facts

10.1 Normaliser for rings

use *order.ML*

method-setup *ring* =
{* *Method.no-args* (*Method.SIMPLE-METHOD'* *HEADGOAL* (*full-simp-tac ring-ss*))
*}
{* *computes distributive normal form in rings* *}

10.2 Rings and the summation operator

lemma *natsum-0* [*simp*]: *setsum f* {..(*0*::*nat*)} = (*f 0*::'*a*::*plus-ac0*)
by *simp*

lemma *natsum-Suc* [*simp*]:
setsum f {..*Suc n*} = (*f* (*Suc n*) + *setsum f* {..*n*}::'*a*::*plus-ac0*)
by (*simp add*: *atMost-Suc*)

lemma *natsum-Suc2*:
setsum f {..*Suc n*} = (*setsum* (%*i*. *f* (*Suc i*)) {..*n*} + *f 0*::'*a*::*plus-ac0*)
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case *Suc* **thus** *?case* **by** (*simp add*: *assoc*)
qed

lemma *natsum-cong* [*cong*]:
!!*k*. [| *j* = *k*; !!*i*::*nat*. *i* <= *k* ==> *f i* = (*g i*::'*a*::*plus-ac0*) |] ==>
setsum f {..*j*} = *setsum g* {..*k*}
by (*induct j*) *auto*

lemma *natsum-zero* [*simp*]: *setsum* (%*i*. *0*) {..*n*::*nat*} = (*0*::'*a*::*plus-ac0*)
by (*induct n*) *simp-all*

lemma *natsum-add* [*simp*]:
!!*f*::*nat*=>'*a*::*plus-ac0*.
setsum (%*i*. *f i* + *g i*) {..*n*::*nat*} = *setsum f* {..*n*} + *setsum g* {..*n*}

```
by (induct n) (simp-all add: plus-ac0)


instance ring < plus-ac0
proof
  fix x y z
  show (x::'a::ring) + y = y + x by (rule a-comm)
  show ((x::'a::ring) + y) + z = x + (y + z) by (rule a-assoc)
  show 0 + (x::'a::ring) = x by (rule l-zero)
qed

ML {*
  local
    val lhss =
        [read-cterm (sign-of (the-context ()))
                    (?t + ?u::'a::ring, TVar (('z, 0), [])),
         read-cterm (sign-of (the-context ()))
                    (?t - ?u::'a::ring, TVar (('z, 0), [])),
         read-cterm (sign-of (the-context ()))
                    (?t * ?u::'a::ring, TVar (('z, 0), [])),
         read-cterm (sign-of (the-context ()))
                    (- ?t::'a::ring, TVar (('z, 0), []))
        ];
    fun proc sg - t =
      let val rew = Tactic.prove sg [] []
            (HOLogic.mk-Trueprop
               (HOLogic.mk-eq (t, Var ((x, Term.maxidx-of-term t + 1), fastype-of
t))))
               (fn - => simp-tac ring-ss 1)
            |> mk-meta-eq;
          val (t', u) = Logic.dest-equals (Thm.prop-of rew);
      in if t' aconv u
        then None
        else Some rew
    end;
  in
    val ring-simproc = mk-simproc ring lhss proc;
  end;
*}

ML-setup {* Addsimprocs [ring-simproc] *}

lemma natsum-ldistr:
  !!a::'a::ring. setsum f {..n::nat} * a = setsum (%i. f i * a) {..n}
by (induct n) simp-all

lemma natsum-rdistr:
  !!a::'a::ring. a * setsum f {..n::nat} = setsum (%i. a * f i) {..n}
```

by (*induct n*) *simp-all*

10.3 Integral Domains

declare *one-not-zero* [*simp*]

lemma *zero-not-one* [*simp*]:
  0 ~= (1::'a::*domain*)
by (*rule not-sym*) *simp*

lemma *integral-iff*:
  ($a * b$ = (0::'a::*domain*)) = (a = 0 | b = 0)
proof
  assume $a * b$ = 0 **then show** a = 0 | b = 0 **by** (*simp add*: *integral*)
next
  assume a = 0 | b = 0 **then show** $a * b$ = 0 **by** *auto*
qed

lemma *m-lcancel*:
  assumes *prem*: (a::'a::*domain*) ~= 0 **shows** *conc*: ($a * b$ = $a * c$) = (b = c)
proof
  assume *eq*: $a * b$ = $a * c$
  then have $a * (b - c)$ = 0 **by** *simp*
  then have a = 0 | $(b - c)$ = 0 **by** (*simp only*: *integral-iff*)
  with *prem* **have** $b - c$ = 0 **by** *auto*
  then have b = $b - (b - c)$ **by** *simp*
  also have $b - (b - c)$ = c **by** *simp*
  finally show b = c **.**
next
  assume b = c **then show** $a * b$ = $a * c$ **by** *simp*
qed

lemma *m-rcancel*:
  (a::'a::*domain*) ~= 0 ==> ($b * a$ = $c * a$) = (b = c)
by (*simp add*: *m-lcancel*)

end

theory *IntegContrib* = *Main* :

11 Auxiliary Arithmetic Lemmas

This theory contains auxiliary lemmas on Isabelle/HOL's *int* that should have been proven in the arithmetic library.

The following lemma is used in JavaIntegersRing:

axioms
pos-le-mod-geq: $[\![(0::int) < b;\ b \leq a]\!] \Longrightarrow a\ mod\ b = (a - b)\ mod\ b$

These lemmas are used in JavaIntegersDiv:

axioms
div-pos-pos-le-pos : $[\![0 \leq a;\ 0 < b]\!] \Longrightarrow (a::int)\ div\ b \leq a$
div-neg-pos-ge-pos : $[\![a < 0;\ 0 < b]\!] \Longrightarrow a \leq (a::int)\ div\ b$
div-notminusone-less : $[\![a < 0;\ b < 0]\!] \Longrightarrow (b::int) \neq -1 \Longrightarrow a\ div\ b < (-a)$
div-pos-pos-gt0 : $[\![b \leq a;\ 0 < a;\ 0 < b]\!] \Longrightarrow (0::int) < a\ div\ b$

lemma *divide-both-sides* :
$(x::int) = y \Longrightarrow x\ div\ a = y\ div\ a$
by *auto*

lemma *div-pos-neg-ge-neg* :
assumes *1*: $0 \leq a$
assumes *2*: $b < 0$
shows $(-a) \leq (a::int)\ div\ b$
apply (*insert 1*)
apply (*cases* $0 < a$)

apply (*subst zminus-zminus* [*symmetric*, *of b*])
apply (*simp only*: *zdiv-zminus2* [*of* $a - b$])
apply (*rule div-neg-pos-ge-pos*)
apply (*simp-all add*: *2*)

apply (*simp add*: *zle-def*[*symmetric*])
apply (*drule zle-anti-sym*)
apply *auto*
done

lemma *div-neg-neg-le-neg* :
assumes *1*: $a < 0$
assumes *2*: $b < 0$
shows $(a::int)\ div\ b \leq (-a)$
apply (*subst zminus-zminus* [*symmetric*, *of b*])
apply (*simp only*: *zdiv-zminus2* [*of* $a - b$])
apply (*rule div-pos-pos-le-pos*)
apply (*simp-all add*: *zless-or-eq-imp-zle 1 2*)
done

lemma *div-neg-neg-gt0*:
$[\![a \leq b;\ a < 0;\ b < 0]\!] \Longrightarrow (0::int) < a\ div\ b$

apply (*simp only*: *zdiv-zminus-zminus*[*symmetric*, *of a b*])
apply (*rule div-pos-pos-gt0*[*of* $-b$ $-a$])
apply *auto*
done

lemma *mult-uminus-shift*:
shows $x * (-\ y) = (-\ x) * (y{::}int)$
by *auto*

lemma *abs-neq0-greater-neq0* :
assumes *1*: $abs\ (x) \leq abs\ (y{::}int)$
assumes *2*: $x \neq 0$
shows $y \neq 0\ \&\ 0 < abs(y)$
apply *simp*
apply (*insert zero-le-zabs*[*of x*])
apply (*insert 1*)
apply (*simp only*: *zabs-def*)
apply (*cases* $x < 0$)
apply (*simp only*: *if-P*)
apply (*cases* $y < 0$)
apply (*simp only*: *if-P*)
apply (*simp only*: *if-not-P if-False*)
apply (*cases* $y < 0$)
apply (*simp only*: *if-P*)
apply (*simp only*: *if-not-P if-False*)
apply (*insert 2*)
apply (*simp only*: *HOL.eq-commute*[*of x 0*])
done

end

theory *JavaIntegersRing* = *JavaIntegersTimes* + *Ring* + *IntegContrib* :

12 Ordering Properties

12.1 Axclass ord

instance *JavaInt* :: *ord* **..**

defs (**overloaded**)
JavaInt-le-def : $(x{::}JavaInt) \leq y \equiv$
$((Rep\text{-}JavaInt\ x) \leq (Rep\text{-}JavaInt\ y))$

JavaInt-less-def : $(x{::}JavaInt) < y \equiv$
$x \leq y\ \&\ \ x \neq y$

Later, in a world with multiple exceptions, this has to be refined:

constdefs
  $geq :: [JavaInt, JavaInt] \Rightarrow bool$
  $geq\ x\ y \equiv y \leq x$

  $greater :: [JavaInt, JavaInt] \Rightarrow bool$
  $greater\ x\ y \equiv y < x$

  $equals :: [JavaInt, JavaInt] \Rightarrow bool$
  $equals\ a\ b \equiv (Rep\text{-}JavaInt\ a = Rep\text{-}JavaInt\ b)$

  $notequals :: [JavaInt, JavaInt] \Rightarrow bool$
  $notequals\ a\ b \equiv \sim (equals\ a\ b)$

12.1.1 ord Lemmas

lemma *MinInt-int-is-least* :
$x : JavaInt \Longrightarrow MinInt\text{-}int \leq x$
by (*simp add*: *JavaInt-def*)

lemma *MinInt-is-least2* :
$MinInt \leq x$
  apply (*simp add*: *MinInt-def*)
  apply (*simp add*: *JavaInt-le-def*)
  apply (*subst Abs-JavaInt-inverse*)
  apply (*simp add*: *MinInt-int-in-JavaInt*)
  apply (*rule MinInt-int-is-least*)
  apply (*rule Rep-JavaInt*)
done

lemma *MaxInt-int-is-largest* :
  $x : JavaInt \Longrightarrow x \leq MaxInt\text{-}int$
  by (*simp add*: *JavaInt-def*)

lemma *MaxInt-is-largest2* :
  $x \leq MaxInt$
  apply (*simp add*: *MaxInt-def*)
  apply (*simp add*: *JavaInt-le-def*)
  apply (*subst Abs-JavaInt-inverse*)
  apply (*simp add*: *MaxInt-int-in-JavaInt*)
  apply (*rule MaxInt-int-is-largest*)
  apply (*rule Rep-JavaInt*)
done

lemma *greater-zero-Abs-Rep*: $(0 < d) = (0 < Rep\text{-}JavaInt\ d)$
  by (*simp add*: *int-less-le JavaInt-less-def JavaInt-le-def zero-def*
  Rep-JavaInt-inject[*symmetric*])

12.2 Axclass order

Lemmas required by the axiomatic class *order* are:

lemma *JavaInt-refl*: $(x::JavaInt) \leq x$
  by (*unfold JavaInt-le-def*, *auto*)

lemma *JavaInt-trans*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq (z::JavaInt)$
 by (*unfold JavaInt-le-def*, *auto*)

lemma *JavaInt-antisym*:
  $(x::JavaInt) \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
apply (*unfold JavaInt-le-def*)
apply (*simp add*: *Rep-JavaInt-inject* [*symmetric*])
done

lemma *JavaInt-less-le*:
$((x::JavaInt) < y) = (x \leq y \ \& \ x \neq y)$
apply (*unfold JavaInt-le-def JavaInt-less-def*)
apply (*simp add*: *Rep-JavaInt-inject* [*symmetric*])
done

instance *JavaInt* :: *order*
apply *intro-classes*
apply (*simp add*: *JavaInt-refl*)
apply (*rule JavaInt-trans*)
apply *auto*
apply (*rule JavaInt-antisym*)
apply *auto*
apply (*simp add*: *JavaInt-less-def*)
apply (*simp add*: *JavaInt-less-def*)
apply (*simp add*: *JavaInt-less-def*)
done

12.3 Axclass linorder

Lemmas required by the axiomatic class *linorder* are:

lemma *JavaInt-linear*:
  $(x::JavaInt) \leq y \mid y \leq x$
  by (*unfold JavaInt-le-def*, *auto*)

instance *JavaInt* :: *linorder*
apply *intro-classes*
apply (*simp add*: *JavaInt-linear*)
done

12.3.1 linorder Lemmas

This is the lowest level of lemmas. It contains characterizing properties of $<$ and $\leq$.

lemma *MinInt-neq-zero* [*simp*]:
  shows *MinInt* $\neq$ *0*
  apply (*simp add*: *MinInt-def zero-def*)
  apply (*subst Abs-JavaInt-inject*)
  apply *simp-all*
  apply (*simp add*: *MinInt-int-def BitLength-def*)
  done

lemma *zero-neq-MinInt* [*simp*]:
  shows *0* $\neq$ *MinInt*
  by (*simp add*: *MinInt-neq-zero*[*symmetric*])

lemma *MaxInt-neq-zero* [*simp*]:
  shows *MaxInt* $\neq$ *0*
  apply (*simp add*: *MaxInt-def zero-def*)
  apply (*subst Abs-JavaInt-inject*)
  apply *simp-all*
  apply (*simp add*: *MaxInt-int-def BitLength-def*)
  done

lemma *zero-neq-MaxInt* [*simp*]:
  shows *0* $\neq$ *MaxInt*
  by (*simp add*: *MaxInt-neq-zero*[*symmetric*])

lemma *MaxInt-neq-MinInt* [*simp*]:
  shows *MaxInt* $\neq$ *MinInt*
  apply (*simp add*: *MaxInt-def MinInt-def*)
  apply (*subst Abs-JavaInt-inject*)
  apply *simp-all*
  apply (*simp add*: *MinInt-int-def MaxInt-int-def BitLength-def*)
  done

lemma *MinInt-neq-MaxInt* [*simp*]:
  shows *MinInt* $\neq$ *MaxInt*
  by (*simp add*: *MaxInt-neq-MinInt*[*symmetric*])

lemma *MinInt-le-zero* [*simp*]:
  shows *MinInt* $<$ *0*
  apply (*simp add*: *HOL.order.order-less-le*)
  apply (*simp add*: *MinInt-is-least2*)
  done

lemma *zero-le-MaxInt* [*simp*]:
  shows *0* $<$ *MaxInt*
  apply (*simp add*: *HOL.order.order-less-le*)

```
  apply (simp add: MaxInt-is-largest2)
  done

lemma MaxMin-compare: MinInt-int < MaxInt-int
  by (simp add: MinInt-int-def MaxInt-int-def BitLength-def)

lemma MinInt-le-MaxInt [simp]:
   shows   MinInt < MaxInt
  apply (simp add: HOL.order.order-less-le)
  apply (simp add: MaxInt-is-largest2)
  done

lemma MinInt-is-least [simp]:
   shows   ~(x < MinInt)
  by (simp add: HOL.linorder-not-less MinInt-is-least2)

lemma MaxInt-is-largest [simp]:
   shows   ~(MaxInt < x)
  by (simp add: HOL.linorder-not-less MaxInt-is-largest2)

lemma Rep-JavaInt-neg-in-JavaInt [simp]:
   assumes 1: MinInt ≠ a
   shows − Rep-JavaInt a : JavaInt
  apply (rule minus-in-JavaInt)
  apply (insert 1)
  apply (simp add: MinInt-def)
  apply (simp add: Rep-JavaInt-inject[of - a, symmetric])
  apply (simp add: Abs-JavaInt-inverse)
  done

lemma neg-less-zero:
   assumes 1: 0 ≤ a
   shows   −(a::JavaInt) ≤ 0
proof −
  have  − Rep-JavaInt a : JavaInt
  apply (rule Rep-JavaInt-neg-in-JavaInt[of a])
  apply (insert 1)
  apply auto
  apply (insert MinInt-le-zero)
  apply (simp only: HOL.linorder-not-less[symmetric])
  apply auto
  done

then show ?thesis
  apply (simp add: uminus-def)
  apply (simp add: Int-to-JavaInt-ident)
  apply (simp add: JavaInt-le-def zero-def)
  apply (simp add: Abs-JavaInt-inverse)
  apply (insert 1)
```

```
  apply (simp add: zero-def JavaInt-le-def)
  done
qed

lemma neq-MinInt-larger-MinInt:
   (a ≠ MinInt) = (MinInt < a)
  apply (simp add: JavaInt-less-def JavaInt-le-def MinInt-def
    Abs-JavaInt-inverse)
  apply (simp add: JavaIntegersDef.eq-commute [of a - ])
  done

lemma larger-zero-neq-MinInt:
  assumes 1: 0 < a
   shows MinInt ≠ a
  apply (insert MinInt-le-zero 1)
  apply (drule HOL.order-less-trans [of MinInt 0 a], simp)
  apply (simp add: neq-MinInt-larger-MinInt[symmetric])
  apply (simp add: JavaIntegersDef.neq-commute[of a MinInt])
  done

lemma uminus-pos-if-neg:
assumes 1: a ≠ MinInt
 assumes 2: a < 0
 shows      0 < − a
  apply (simp add: uminus-def)
  apply (insert 1)
  apply (simp add: JavaIntegersDef.neq-commute[of a MinInt])
  apply (simp add: Int-to-JavaInt-ident[of − Rep-JavaInt a]
    Rep-JavaInt-neg-in-JavaInt[of a])
  apply (insert 2)
  apply (simp add: JavaInt-le-def JavaInt-less-def zero-def
    Abs-JavaInt-inverse)
  apply (simp add: Rep-JavaInt-inject[symmetric] Abs-JavaInt-inverse)
  done

lemma uminus-neg-if-pos:
 assumes 1: 0 < (a::JavaInt)
 shows      −a < 0
  apply (simp add: uminus-def)
  apply (insert 1)
  apply (frule larger-zero-neq-MinInt)
  apply (simp add: Int-to-JavaInt-ident[of − Rep-JavaInt a]
    Rep-JavaInt-neg-in-JavaInt[of a])
  apply (simp add: JavaInt-le-def JavaInt-less-def zero-def)
  apply (simp add: Rep-JavaInt-inject[symmetric] Abs-JavaInt-inverse)
  done

lemma uminus-neg1:
  assumes 1: − b < 0
```

```
  assumes 2: b ≠ MinInt
    shows 0 < (b::JavaInt)
  apply (insert 1 2)
  apply (simp add: JavaInt-less-def JavaInt-le-def uminus-def zero-def
  Rep-JavaInt-inject[symmetric] MinInt-def Abs-JavaInt-inverse)
  apply (simp add: Int-to-JavaInt-ident [of − Rep-JavaInt b]
    minus-in-JavaInt Abs-JavaInt-inverse)
done

lemma uminus-neg2:
  assumes 1:  0 < − b
  assumes 2: b ≠ MinInt
    shows (b::JavaInt) < 0
  apply (insert 1 2)
  apply (simp add: JavaInt-less-def JavaInt-le-def uminus-def zero-def
  Rep-JavaInt-inject[symmetric] MinInt-def)
  apply (simp add: Int-to-JavaInt-ident [of − Rep-JavaInt b]
    minus-in-JavaInt Abs-JavaInt-inverse)
done
```

The following lemma is highly important. It relates the < operation of *int* and *JavaInt*.

```
lemma le-quasi-def:
   shows (a < b) = (Rep-JavaInt a < Rep-JavaInt b)
  apply (simp add: JavaInt-less-def JavaInt-le-def)
  apply (simp add: Rep-JavaInt-inject[of a b, symmetric])
  apply (simp add: int-less-le[symmetric])
  done


lemma le-neg-bound : xa <= − x ==> Rep-JavaInt xa <= − Rep-JavaInt x
  apply (cases x = MinInt)
  prefer 2
  apply (subst uminus-homom[symmetric])
  prefer 2
  apply (simp-all add: JavaIntegersRing.JavaInt-le-def uminus-MinInt )
  apply (simp add: JavaIntegersRing.JavaInt-le-def[symmetric])
  apply (simp add: order-le-less )
  apply (rule disjI1)
  apply (simp add: JavaIntegersRing.JavaInt-le-def JavaIntegersAdd.uminus-def
                MinInt-def )
  apply (simp add: MinInt-int-def BitLength-def  )
  apply (subst Abs-JavaInt-inverse)
  prefer 2
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def )
  done
```

lemma *less-vs-not-le*:
  shows $((z :: JavaInt) \leq w) = (\sim(w < z))$
 apply (*simp add*: *HOL.order.order-less-le*)
 apply *auto*
 apply (*simp add*: *HOL.order.order-antisym*)
 apply (*simp add*: *JavaInt-le-def*)
 done

The following lemma is highly important.

lemma *less-vs-eq-or-le*:
  shows $((z{::}JavaInt) \leq w) = ((z = w) \mid (z < w))$
 apply (*simp add*: *JavaInt-less-def*)
 apply *auto*
 done

lemma *less-greater-contradict*:
 assumes *1*: $x < y$
 assumes *2*: $y < (x{::}JavaInt)$
 shows *False*
 apply (*insert 1 2*)
 apply (*simp add*: *JavaInt-less-def JavaInt-le-def Rep-JavaInt-inject*[*symmetric*])
 apply *auto*
 done

lemma *neg-less-neg-greater-eq*:
 assumes *1*: $\sim (x < y)$
 assumes *2*: $\sim (y < (x{::}JavaInt))$
 shows $x = y$
 apply (*insert 1 2*)
 apply (*simp add*: *JavaInt-less-def JavaInt-le-def Rep-JavaInt-inject*[*symmetric*])
 apply *auto*
 done

lemma *leq-geq-eq*:
 assumes *1*: $b \leq (a{::}JavaInt)$
 assumes *2*: $a \leq b$
 shows $a = b$
 apply (*insert 1 2*)
 apply (*simp add*: *JavaInt-le-def Rep-JavaInt-inject*[*symmetric*])
 done

lemma *pos-neg-eq*:
 assumes *1*: $a \leq 0$
 assumes *2*: $- a \leq 0$
 assumes *3*: $a \neq MinInt$
 shows $a = (0{::}JavaInt)$
 apply (*insert 1 2 3*)
 apply (*simp add*: *JavaInt-le-def zero-def uminus-def*)

apply (*simp add*: *Rep-JavaInt-inject*[*symmetric*] *MinInt-def*
Abs-JavaInt-inverse)
apply (*simp add*: *Int-to-JavaInt-ident minus-in-JavaInt Abs-JavaInt-inverse*)
done

The following lemma is highly important.

lemma *weaken-less*:
assumes *1*: $a < b$
shows $(a::JavaInt) \leq b$
apply (*insert 1*)
apply (*simp add*: *JavaInt-less-def*)
done

lemma *strengthen-less*:
assumes *1* : $a \neq b$
assumes *2* : $a \leq b$
shows $(a::JavaInt) < b$
apply (*insert 1 2*)
apply (*simp add*: *less-vs-eq-or-le*)
done

lemma *swap-le*:
assumes *1*: $a \neq MinInt$
assumes *2*: $b \neq MinInt$
shows $(-a < -b) = (b < a)$
apply (*simp add*: *uminus-def*)
apply (*insert 1 2*)
apply (*simp only*: *HOL.neq-commute*)
apply (*simp add*: *Int-to-JavaInt-ident*)
apply (*simp add*: *JavaInt-less-def* [*of Abs-JavaInt* (− *Rep-JavaInt a*) -] *JavaInt-le-def*)
apply (*simp add*: *Abs-JavaInt-inverse*)
apply (*simp add*: *Abs-JavaInt-inject*)
apply (*simp add*: *zminus-equation* [*of Rep-JavaInt a* − *Rep-JavaInt b*])
apply (*simp add*: *JavaInt-less-def JavaInt-le-def*)
apply (*simp add*: *Rep-JavaInt-inject*)
done

12.4 The Absolute Value Function `abs`

We also need to model the absolute value function on JavaInt (in Java, this is provided by the method `public static int java.lang.Math.abs(int a)`):

defs (**overloaded**)
abs-def : *abs* $(x::JavaInt) \equiv$ *Int-to-JavaInt* (*abs* (*Rep-JavaInt x*))

This function has an unexpected property, which is given in the Java API

Specification[2]:

Java API Specification [Jav]

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.
Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable int value, the result is that same value, which is negative.

12.4.1 `abs` Lemmas

To formalize these quite surprising properties, we introduce the following lemmas. They characterize the absolute value function `abs` which was defined above.

lemma *abs-pos*:
  assumes *1*: $0 \leq a$
  shows $abs(a::JavaInt) = a$
  apply (*insert 1*)
  apply (*simp add*: *abs-def zabs-def JavaInt-le-def*
    zle-def zero-def Abs-Rep-Mod-identity-abstract)
  done

lemma *abs-neg*:
  assumes *1*: $a \leq 0$
  shows $abs\ (a::JavaInt) = -\ a$
  apply (*insert 1*)
  apply (*simp add*: *abs-def zabs-def JavaInt-less-def JavaInt-le-def*
    zle-def zero-def uminus-def)
  apply (*auto*)
  apply (*simp add*: *zle-def* [*symmetric*])
  apply (*tactic dtac* (*int-le-less RS iffD1*) *1*)
  apply (*tactic dtac* (*int-le-less RS iffD1*) *1*)
  apply (*auto*)
  done

lemma *abs-MinInt* [*simp*]:
  shows *abs MinInt* = *MinInt*
  apply (*subst abs-neg*)
  apply (*auto simp*: *order-le-less intro*: *uminus-MinInt MinInt-le-zero*)
  done

lemma *abs-MaxInt* [*simp*]:
  shows *abs MaxInt* = *MaxInt*
  apply (*subst abs-pos*)

[2] http://java.sun.com/j2se/1.4.1/docs/api/java/lang/Math.html#abs(int)

```
  apply (auto intro:weaken-less)
  done

lemma abs-zero [simp]:
  shows abs (0::JavaInt) = 0
  apply (subst abs-pos)
  apply (auto)
  done


lemma abs-positive:
  assumes 1: x ≠ MinInt
  shows     0 ≤ abs (x::JavaInt)
  apply (cases 0≤x)
  apply (simp add: abs-pos)
  apply (insert 1)
  apply (subst abs-neg)
  apply (simp add: abs-neg less-vs-not-le)
  apply (simp add: less-vs-not-le[symmetric] less-vs-eq-or-le)
  apply (simp add: linorder-not-le)
  apply (cases − x = 0)
  apply auto
  apply (simp-all add: order-le-less)

  apply (cut-tac x = x in MinInt-is-least2)
  apply (rule uminus-pos-if-neg)
  apply (auto simp: order-le-less)
  done
```

12.5 Formalizing More plus Lemmas

This section contains more lemmas that arise from the Java Language Specification § 15.18.2. They had to be postponed to here because they require comparison operators which had not been formalized in Sec. 5.1.1 yet.

```
lemma Plus-leq-2MaxInt:
 Rep-JavaInt a + Rep-JavaInt b ≤ 2 ∗ MaxInt-int
  proof −
    have ∗ : Rep-JavaInt a + Rep-JavaInt b ≤ MaxInt-int + Rep-JavaInt b
      by (rule zadd-zle-mono1, simp add: MaxInt-geq )
    moreover
    have ∗∗ : ... ≤ MaxInt-int + MaxInt-int
      by (rule zadd-zle-mono2, simp add: MaxInt-geq )
    moreover have ∗∗∗ : ... = 2 ∗ MaxInt-int
      by auto
    ultimately show ?thesis
      by (subst ∗∗∗ [symmetric], blast intro: order-trans)
  qed
```

```
lemma Plus-geq-2MinInt:
  2 ∗ MinInt-int ≤ Rep-JavaInt a + Rep-JavaInt b
    proof −
      have ∗ : MinInt-int + Rep-JavaInt b ≤ Rep-JavaInt a + Rep-JavaInt b
        by (rule zadd-zle-mono1, simp add: MinInt-leq )
      moreover
      have ∗∗ : MinInt-int + MinInt-int ≤ MinInt-int + Rep-JavaInt b
        by (rule zadd-zle-mono2, simp add: MinInt-leq )
      moreover have ∗∗∗ : MinInt-int + MinInt-int = 2 ∗ MinInt-int
        by auto
      ultimately show ?thesis
        by (subst ∗∗∗ [symmetric], blast intro: order-trans )
    qed
```

This lemma captures the property described in (4) in the JLS, §15.18.2:

```
lemma JavaInt-add-overflow-sign :
  assumes 1: MaxInt-int < Rep-JavaInt a + Rep-JavaInt b
  shows a + b < 0
  proof −

have upperbound: Rep-JavaInt a + Rep-JavaInt b − MinInt-int + (2 ∗ MinInt-int)

  < − (2 ∗ MinInt-int)
  apply (simp add: JavaInt-maxmin-compare2)
  apply (cut-tac a = a and b = b in Plus-leq-2MaxInt)
  apply (simp only: zle-add1-eq-le [symmetric])
  apply (subgoal-tac 2 ∗ MaxInt-int + 1 < 3 + 3 ∗ MaxInt-int)
  apply (blast intro: zless-trans)
  apply (simp add: MaxInt-int-def BitLength-def)
  done

also
have lowerbound: 0 ≤ MinInt-int + (Rep-JavaInt a + Rep-JavaInt b)
  apply (simp only: JavaInt-maxmin-compare2)
   apply (simp only: zadd-commute [of − (MaxInt-int + 1) Rep-JavaInt a +
Rep-JavaInt b])
  apply (simp only: zdiff-def [symmetric])
 apply (simp add: zle-zdiff-eq [of 0 Rep-JavaInt a + Rep-JavaInt b (MaxInt-int+1)])
  apply (simp only: add1-zle-eq 1)
done

also
have A1: 0 < − (2 ∗ MinInt-int)
  by (simp add: MinInt-int-def BitLength-def)

moreover
have A2: MinInt-int + (Rep-JavaInt a + Rep-JavaInt b) < − (2 ∗ MinInt-int)
  apply (simp add: upperbound)
  apply (simp only: zadd-commute [of 3 ∗ MinInt-int -])
```

```
  apply (simp only: zless-zdiff-eq [of - - 3 * MinInt-int, symmetric])
  apply (simp add: JavaInt-maxmin-compare2)
  apply (simp add: add1-zle-eq [symmetric])
  apply (cut-tac a = a and b = b in Plus-leq-2MaxInt)
  apply (subgoal-tac 2 * MaxInt-int ≤ 2 + 3 * MaxInt-int)
  apply (blast intro: order-trans)

  apply (simp add: MaxInt-int-def BitLength-def)
done

moreover
have A3: 2 * MinInt-int + (Rep-JavaInt a + Rep-JavaInt b) < 0
  apply (simp only: zadd-commute [of 2 * MinInt-int -])
  apply (simp only: zless-zdiff-eq [of - - 2 * MinInt-int, symmetric])
  apply (simp add: JavaInt-maxmin-compare2)
  apply (cut-tac a = a and b = b in Plus-leq-2MaxInt)
  apply (simp only: zle-add1-eq-le [symmetric])
done

moreover
have mainproof: (Rep-JavaInt a + Rep-JavaInt b − MinInt-int) mod − (2 *
MinInt-int)
  + MinInt-int < 0
  apply (simp add: zless-zdiff-eq [symmetric])
  apply (simp add: pos-le-mod-geq A1 lowerbound)
  apply (simp add: mod-pos-pos-trivial A2 lowerbound)
  apply (simp add: A3)
  done

ultimately
  show ?thesis
    apply (insert 1)
  apply (simp add: add-def JavaInt-less-def JavaInt-le-def)
  apply (simp add: zero-def Rep-JavaInt-inject [symmetric])
  apply (simp add: Int-to-JavaInt-def)
  apply (simp add: Rep-Abs-remove)
  apply (insert mainproof)
  apply auto
  done
qed
```

This is a good example of how inexact several parts of the Java Language Specification are. If indeed only overflow, i.e. regarding two operands whose sum is larger than MaxInt, is meant here, then why pose such a complicated question? "the sign of the mathematical sum of the two operand values" will always be positive in this case, so why talk about "the sign of the result is not the same"? It would be much clearer to state "the sign of the result is always negative". But what if the authors also wanted to describe underflow, i.e. negative overflow, which is sometimes also referred

to as "overflow"? In §4.2.2 the JLS states "The built-in integer operators do not indicate overflow or underflow in any way." Thus, the term "underflow" is known to the authors and is used in the JLS. Why do they not use it in the context quoted above? This would also explain the complicated phrasing of the above formulation.

To clarify these matters, we add the lemma

```
lemma JavaInt-add-underflow-sign :
  assumes 1: Rep-JavaInt a + Rep-JavaInt b < MinInt-int
  shows 0 ≤ a + b
proof −

have A1:
  0 ≤ Rep-JavaInt a + Rep-JavaInt b − MinInt-int + − (2 ∗ MinInt-int)
  apply simp
 apply (simp only: zadd-commute [of −3 ∗ MinInt-int Rep-JavaInt a + Rep-JavaInt
b])
 apply (simp only: zdiff-zle-eq [of - - Rep-JavaInt a + Rep-JavaInt b, symmetric])
  apply simp
  apply (cut-tac a = a and b = b in Plus-geq-2MinInt)
  apply (subgoal-tac − (−3 ∗ MinInt-int) ≤ 2 ∗ MinInt-int)
  apply (blast intro: order-trans)

  apply (simp add: MinInt-int-def BitLength-def)
  done

also
have A2:
  Rep-JavaInt a + Rep-JavaInt b − MinInt-int + − (2 ∗ MinInt-int) < − (2 ∗
MinInt-int)
  by (simp add: 1)

also
have mainproof:
  0 ≤ (Rep-JavaInt a + Rep-JavaInt b − MinInt-int) mod − (2 ∗ MinInt-int) +
          MinInt-int
  apply (simp add: zdiff-zle-eq [symmetric])
 apply (simp only: zmod-zadd-self2 [of Rep-JavaInt a + Rep-JavaInt b − MinInt-int
   − (2 ∗ MinInt-int), symmetric])
  apply (simp only: mod-pos-pos-trivial A1 A2)
  apply (simp add: Plus-geq-2MinInt)
  done

show ?thesis
  apply (insert 1)
  apply (simp add: add-def zero-def JavaInt-le-def JavaInt-less-def)
  apply (simp add: Int-to-JavaInt-def)
  apply (simp add: Rep-Abs-remove)
  apply (simp only: mainproof)
```

done
qed

13 Ring Properties

Now we can show that normal behaviour JavaIntegers have exactly ring properties. There is a slight confusion here, since the Isabelle2003 Ring class requires in fact field properties (like "there exists a multiplicative inverse", but allows to fake them). This will be cleaned up if ported to a more recent Isabelle version.

13.1 Axclass power

instance *JavaInt* :: *power* **..**

defs (**overloaded**)
power-def : (*a*::*JavaInt*) ^ *n* ≡ *nat-rec 1* (λ*u b*. *b* ∗ *a*) *n*

13.2 Axclass inverse

instance *JavaInt* :: *inverse* **..**

defs (**overloaded**)
inverse-def : *inverse* (*a*::*JavaInt*) ≡ (*if a dvd 1 then THE x*. *a*∗*x* = *1 else 0*)

divide-def : *divide* (*a*::*JavaInt*) *b* ≡ *a* ∗ *inverse b*

13.3 Axclass ring

instance *JavaInt* :: *ring*
apply *intro-classes*
apply (*rule JavaInt-add-assoc*)
apply (*rule JavaInt-add-left-neutral*)
apply (*rule JavaInt-add-minus-inverse*)
apply (*rule JavaInt-add-commute*)
apply (*rule JavaInt-times-assoc*)
apply (*rule JavaInt-times-one-ident*)
apply (*rule JavaInt-add-times-left-distrib*)
apply (*rule JavaInt-times-commute*)
apply (*rule diff-uminus*)
apply (*simp add*: *inverse-def one-def*)
apply (*simp add*: *divide-def*)
apply (*simp add*: *power-def one-def*)
done

end

theory *JavaIntegersDiv* = *JavaIntegersRing* :

14 Division Operator

14.1 Division Definition

instance *JavaInt* :: *Divides.div* **..**

In Java, the division operator comes as a surprise if compared to the mathematical definition of division, which is also used in Isabelle/HOL:

Java Language Specification [GJSB00], §15.17.2

"The binary / operator performs division, producing the quotient of its operands. [...] Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \times q| \leq |n|$; moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.(1)
There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend.(2)
Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an ArithmeticException is thrown.(3)"

This definition points out a major difference between the definition of division in Isabelle/HOL and Java. If the signs of dividend and divisor are different, the results differ by one because Java rounds towards 0 whereas Isabelle/HOL floors the result. Thus, the naïve approach of modelling Java integers by partialization of the corresponding operations of a theorem prover gives the wrong results in these cases.

We model the division by performing case distinctions:

defs (**overloaded**)
div-def : (*x*::*JavaInt*) *div y* ≡
if ((*0* < *x* & *y* < *0*) | (*x* < *0* & *0* < *y*)) & ~ (*EX z*. *Rep-JavaInt x* = *z* ∗ *Rep-JavaInt y*)
then
Int-to-JavaInt (*Rep-JavaInt x div Rep-JavaInt y*) + *1*
else
Int-to-JavaInt (*Rep-JavaInt x div Rep-JavaInt y*)

In a first version of the formalization, we omitted the second condition, namely that the result of Isabelle's division operator is only to be increased

by 1 if y is not a factor of x. We are thankful to Bart Jacobs who pointed us to this flaw (and to the same flaw in the modulo formalization, see below).

The properties mentioned in the language report are formalized below.

Property (3) is not modelled by the theory presented in this section because this theory does not introduce a bottom element for integers in order to treat exceptional cases. Our model returns 0 in this case due to the definition of the total function div in Isabelle/HOL. Exceptions are handled by the next theory layer (see Sec. 18) which adds a bottom element to JavaInt and lifts all operations in order to treat exceptions appropriately.

14.2 Division Representation Boundedness

```
declare Abs-JavaInt-inverse [simp]

lemma div-in-JavaInt :
  assumes A: Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1
  shows Rep-JavaInt a div Rep-JavaInt b ∈ JavaInt
proof −
  {
    assume 1: 0 ≤ Rep-JavaInt a
    assume 2: Rep-JavaInt b < 0
    have Rep-JavaInt a div Rep-JavaInt b ∈ JavaInt
      apply (simp add: JavaInt-def)
      apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
        in div-pos-neg-ge-neg)
      apply (simp-all add: 1 2)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b in div-nonneg-neg-le0)
      apply (cut-tac 1, simp)
      apply (cut-tac 2, simp)
      apply (insert MinInt-leq-minus[of a]  null-leq-MaxInt )
      apply (blast intro: order-trans)
      done
  } note div-in-JavaInt-pos-neg = this

  also
  {
    assume 1: 0 ≤ Rep-JavaInt a
    assume 2: 0 < Rep-JavaInt b
    have Rep-JavaInt a div Rep-JavaInt b ∈ JavaInt
      apply (simp add: JavaInt-def)
      apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b in
        div-pos-pos-le-pos)
      apply (simp-all add: 1 2)
      apply (insert pos-imp-zdiv-nonneg-iff[of Rep-JavaInt b Rep-JavaInt a]
        MinInt-leq0 MaxInt-geq[of a] 1 2)
      apply (blast intro: order-trans)
      done
  } note div-in-JavaInt-nonneg-pos = this
```

```
moreover
{
  assume 1: Rep-JavaInt a < 0
  assume 2: 0 < Rep-JavaInt b
  have Rep-JavaInt a div Rep-JavaInt b ∈ JavaInt
    apply (simp add: JavaInt-def)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b in
      div-neg-pos-ge-pos)
    apply (simp-all add: 1 2)
    apply (insert pos-imp-zdiv-neg-iff[of Rep-JavaInt b Rep-JavaInt a]
      MinInt-leq[of a] null-leq-MaxInt 1 2)
    apply (blast intro: order-trans disjI1 [THEN zless-or-eq-imp-zle])
    done
} note div-in-JavaInt-neg-pos = this

moreover
{
  assume 1: Rep-JavaInt a < 0
  assume 2: Rep-JavaInt b < 0
  assume 3: Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1
  have Rep-JavaInt a div Rep-JavaInt b ∈ JavaInt
    apply (simp add: JavaInt-def)
   apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b in neg-imp-zdiv-nonneg-iff)
    apply (simp add: 2)
    apply (insert 1, rotate-tac)
    apply (drule HOL.order-less-imp-le, simp)
    apply (insert MinInt-leq0)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in div-neg-neg-le-neg)
    apply (simp-all add: 1 2)
    apply (cases Rep-JavaInt a ≠ MinInt-int)
    apply (insert Not-MinInt-minus-leq-MaxInt[of a] MinInt-leq0)
    apply (blast intro: order-trans)

    apply (insert 3, simp)
   apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b in div-notminusone-less)
    apply (simp-all add: 1 2)
    apply (simp add: JavaInt-maxmin-compare[symmetric])
    done
} note div-in-JavaInt-neg-neg = this

show ?thesis
  apply (cases 0 ≤ Rep-JavaInt a)
  apply (cases Rep-JavaInt b < 0)
  apply (simp add: div-in-JavaInt-pos-neg)
  apply (cases Rep-JavaInt b = 0)
  apply (simp add: DIVISION-BY-ZERO zero-in-JavaInt)
  apply (rotate-tac −2)
```

```
    apply (simp add: zle-def [symmetric])
    apply (drule zle-imp-zless-or-eq)
    apply (simp add: div-in-JavaInt-nonneg-pos)

    apply (simp add: zle-def)
    apply (cases 0 < Rep-JavaInt b)
    apply (simp add: div-in-JavaInt-neg-pos)
    apply (cases Rep-JavaInt b = 0)
    apply (simp add: DIVISION-BY-ZERO zero-in-JavaInt)
    apply (rotate-tac −2)
    apply (simp add: zle-def [symmetric])
    apply (drule zle-imp-zless-or-eq)
    apply simp
    apply (insert A)
    apply (simp add: div-in-JavaInt-neg-neg)
    done
qed

lemma JavaInt-nondvd-neq-minusone:
  assumes 1: Rep-JavaInt a mod  Rep-JavaInt b ≠ 0
  shows      b ≠ −1
proof −
  show ?thesis
  apply (insert 1, simp add: number-of-def)
  apply (subst Int-to-JavaInt-ident, rule minusone-in-JavaInt)
  apply (erule swap)
  apply (simp only:Abs-JavaInt-inverse minusone-in-JavaInt
                zmod-minus1-right not-not)
  done
qed


lemma JavaInt-nondvd-div-in-JavaInt:
  assumes 1: Rep-JavaInt a mod  Rep-JavaInt b ≠ 0
  shows      Rep-JavaInt a div  Rep-JavaInt b ∈ JavaInt
proof −
show ?thesis
  apply (rule div-in-JavaInt)
  apply (insert 1)
  apply (drule JavaInt-nondvd-neq-minusone)
  apply (rule disjI2)
  apply (erule swap)
  apply (simp add: number-of-def)
  apply (subst Int-to-JavaInt-ident, rule minusone-in-JavaInt)
  apply (subst Rep-JavaInt-inject[symmetric])
  apply (simp only:Abs-JavaInt-inverse minusone-in-JavaInt)
  done
qed
```

14.3 Division Characterizations

lemma *JavaInt-div-nondvd-pos-pos*:
assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* $\neq$*0*
assumes *2*: *0* $<$ *a*
assumes *3*: *0* $<$ *b*
shows ((*a*::*JavaInt*) *div b*)=
Abs-JavaInt (*Rep-JavaInt a div Rep-JavaInt b*)
proof −
show *?thesis*
apply (*insert 1*,*insert 2*,*insert 3*)
apply (*simp add*:*div-def split del*: *split-if*)
apply (*subst if-not-P*)
apply (*simp-all add*: *Int-to-JavaInt-ident*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*rule order-less-not-sym*)
defer *1*
apply (*rule order-less-not-sym*)
apply (*simp-all add*: *Int-to-JavaInt-ident JavaInt-nondvd-div-in-JavaInt*)
done
qed

lemma *JavaInt-div-nondvd-neg-neg*:
assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* $\neq$*0*
assumes *2*: *a* $<$ *0*
assumes *3*: *b* $<$ *0*
shows ((*a*::*JavaInt*) *div b*)= *Abs-JavaInt* (*Rep-JavaInt a div Rep-JavaInt b*)
proof −
show *?thesis*
apply (*insert 1*,*insert 2*,*insert 3*)
apply (*simp add*:*div-def split del*: *split-if*)
apply (*subst if-not-P*)
apply (*simp-all add*: *Int-to-JavaInt-ident*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*rule order-less-not-sym*)
defer *1*
apply (*rule order-less-not-sym*)
apply (*simp-all add*: *Int-to-JavaInt-ident JavaInt-nondvd-div-in-JavaInt*)
done
qed

lemma *JavaInt-div-nondvd-pos-neg*:
assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* $\neq$*0*
assumes *2*: *0* $<$ *a*
assumes *3*: *b* $<$ *0*
shows ((*a*::*JavaInt*) *div b*)= *Abs-JavaInt* (*Rep-JavaInt a div Rep-JavaInt b*) + *1*

```
proof −
show ?thesis
  apply (insert 1,insert 2,insert 3)
  apply (simp add:div-def split del: split-if)
  apply (subst if-P)
  apply (simp ! add: zmod-eq-0-iff zero-def)
  apply auto
  apply (simp-all add: Int-to-JavaInt-ident JavaInt-nondvd-div-in-JavaInt)
  done
qed


lemma JavaInt-div-nondvd-neg-pos:
  assumes 1: Rep-JavaInt a mod  Rep-JavaInt b ≠0
  assumes 2: a < 0
  assumes 3: 0 < b
  shows     ((a::JavaInt) div b)= Abs-JavaInt (Rep-JavaInt a div Rep-JavaInt
b) + 1
proof −
show ?thesis
  apply (insert 1,insert 2,insert 3)
  apply (simp add:div-def split del: split-if)
  apply (subst if-P)
  apply (simp-all add: Int-to-JavaInt-ident)

  apply (simp ! add: zmod-eq-0-iff zero-def)
  apply auto
  apply (simp-all add: Int-to-JavaInt-ident JavaInt-nondvd-div-in-JavaInt)
  done
qed
```

15 Remainder Operator

15.1 Remainder Definition

The remainder operator is closely related to the division operator. Thus, it does not conform to standard mathematical definitions either.

Java Language Specification [GJSB00], §15.17.3

"The binary % operator is said to yield the remainder of its operands from an implied division [...] The remainder operation for operands that are integers after binary numeric promotion (§ 5.6.2) produces a result value such that $(a/b) * b + (a\%b)$ is equal to a.(1)
This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0).(2)
It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive;(3)
moreover, the magnitude of the result is always less than the magnitude of the divisor.(4)
If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.(5)

Examples:	5%3 produces 2	(note that 5/3 produces 1)
	5%(-3) produces 2	(note that 5/(-3) produces -1)
	(-5)%3 produces -2	(note that (-5)/3 produces -1)
	(-5)%(-3) produces -2	(note that (-5)/(-3) produces 1)(6)"

When formalizing the remainder operator, we have to keep in mind the formalization of the division operator and the required equality (1). Therefore, the remainder operator mod is formalized as follows:

defs (**overloaded**)
mod-def : (*x*::*JavaInt*) *mod y* ≡
if ((*0* < *x* & *y* < *0*) | (*x* < *0* & *0* < *y*)) & ~ (*EX z*. *Rep-JavaInt x* = *z* ∗ *Rep-JavaInt y*)
then
Int-to-JavaInt (*Rep-JavaInt x mod Rep-JavaInt y*) − *y*
else
Int-to-JavaInt (*Rep-JavaInt x mod Rep-JavaInt y*)

For property (5), see the discussion of the division operator above.

Again, it is not made explicit in the JLS what happens if the dividend of a modulo operation equals 0.

Java is not the only language whose definitions of div and mod do not resemble the mathematical definitions. The languages Fortran, Pascal and Ada define division in the same way as Java, and Fortran's MOD and Ada's REM operators are modelled in the same way as Java's % operator. Goldberg [Gol02, p. H-12] expresses his regret for these differences and argues in favour of the mathematical definition.

15.2 Remainder Representation Boundedness

lemma *mod-in-JavaInt* [*simp*] : *Rep-JavaInt a mod Rep-JavaInt b* ∈ *JavaInt*
proof −
have *Rep-JavaInt b* < *0* ⟹ *Rep-JavaInt a mod Rep-JavaInt b* ∈ *JavaInt*

```
    apply (simp add: JavaInt-def)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in IntDiv.neg-mod-bound, simp)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in IntDiv.neg-mod-sign, simp)
    apply (rotate-tac)
    apply (drule disjI1 [THEN zless-or-eq-imp-zle])
    apply (cut-tac b = b in MinInt-leq)
    apply (blast intro: order-trans null-leq-MaxInt)
    done

  also have
    0 < Rep-JavaInt b ⟹ Rep-JavaInt a mod Rep-JavaInt b ∈ JavaInt
    apply (simp add: JavaInt-def)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in pos-mod-bound, simp)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in pos-mod-sign, simp)
    apply (cut-tac b = b in MaxInt-geq)
    apply (rotate-tac)
    apply (drule disjI1 [THEN zless-or-eq-imp-zle])
    apply (blast intro: order-trans MinInt-leq0)
    done

  ultimately show ?thesis
    apply (cases Rep-JavaInt b < 0)
    apply simp
    apply (cases Rep-JavaInt b = 0)
    apply (simp add: DIVISION-BY-ZERO Rep-JavaInt)
    apply (simp add: int-neq-iff)
    done
qed


lemma mod-minus-divisor-in-JavaInt [simp] :
 assumes 1: 0 < Rep-JavaInt b
 shows − Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b ∈ JavaInt

proof −

  have HL3a: 0 < MaxInt-int
    by (simp add: MaxInt-int-def BitLength-def)

{
  assume 1: 0 < Rep-JavaInt b
  have MinInt-int < − Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b
    apply (simp only: zadd-commute[of − Rep-JavaInt b -])
    apply (simp only: zdiff-zless-eq[symmetric, of MinInt-int - − Rep-JavaInt b])
    apply (simp )
```

```
    apply (insert MaxInt-geq [of b])
    apply (drule zadd-zle-mono1 [of Rep-JavaInt b MaxInt-int MinInt-int])
    apply (simp add: MaxInt-MinInt-add)
     apply (simp add: zle-add1-eq-le [symmetric, of Rep-JavaInt b + MinInt-int
−1])
    apply (insert 1)
    apply (drule pos-mod-sign [of Rep-JavaInt b Rep-JavaInt a])
    apply simp
    done
} note HL1 = this

{
  assume 1: 0 < Rep-JavaInt b
  have − Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b ≤ MaxInt-int
    apply (simp only: zadd-commute[of − Rep-JavaInt b -])
    apply (simp only: zdiff-def[symmetric])
    apply (simp only: zdiff-zle-eq[of - Rep-JavaInt b MaxInt-int])
    apply (insert 1)
    apply (drule pos-mod-bound[of Rep-JavaInt b Rep-JavaInt a])
    apply (insert HL3a)
    apply (drule order-less-imp-le)
    apply (drule zadd-zless-mono [of 0 MaxInt-int Rep-JavaInt a mod Rep-JavaInt
b Rep-JavaInt b], simp)
    apply (drule order-less-imp-le, simp)
    done
} note HL3 = this

show ?thesis
   apply (simp add: JavaInt-def)
   apply (rule conjI)
   apply (insert MinInt-leq0 1)
   apply (drule HL1)
   apply (simp only: order-less-imp-le)
   apply (rule HL3, simp)
   done
qed

lemma mod-minus-in-JavaInt :
  assumes 2: Rep-JavaInt b ≠ MinInt-int
  shows Rep-JavaInt a mod Rep-JavaInt b − Rep-JavaInt b ∈ JavaInt
  proof −

  {
    assume 1: 0 < Rep-JavaInt b
    have Rep-JavaInt a mod Rep-JavaInt b − Rep-JavaInt b ∈ JavaInt
      apply (simp add: JavaInt-def)
      apply auto
      apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
        in pos-mod-sign, simp add: 1)
```

```
  apply (simp add: zle-zdiff-eq)
  apply (subgoal-tac MinInt-int + Rep-JavaInt b ≤ 0)
  apply (blast intro: order-trans)

  apply (cut-tac b = b in MaxInt-geq)
  apply (subgoal-tac MinInt-int + Rep-JavaInt b ≤ MinInt-int + MaxInt-int)
  apply (subgoal-tac MinInt-int + MaxInt-int ≤ 0)
  apply (blast intro: order-trans)
  apply (simp add: MinInt-int-def MaxInt-int-def BitLength-def)
  apply (rule zadd-zle-mono2, simp)

  apply (simp add: zdiff-zle-eq)
  apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
    in pos-mod-bound, simp add: 1)
  apply (rotate-tac −1)
  apply (drule int-less-le [THEN iffD1])
  apply (subgoal-tac Rep-JavaInt b ≤ MaxInt-int + Rep-JavaInt b)
  apply (blast intro: order-trans)
  apply (simp add: MaxInt-int-def BitLength-def)
  done
} note mod-minus-in-JavaInt1 = this

moreover
{
  assume 1: Rep-JavaInt b < 0
  have Rep-JavaInt a mod Rep-JavaInt b − Rep-JavaInt b ∈ JavaInt
    apply (simp add: JavaInt-def)
    apply auto
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in IntDiv.neg-mod-bound, simp add: 1)
    apply (drule int-less-le [THEN iffD1])
    apply (simp add: zle-zdiff-eq)
    apply auto
    apply (subgoal-tac MinInt-int + Rep-JavaInt b ≤ MinInt-int)
    apply (cut-tac b = b in MinInt-leq)
    apply (blast intro: order-trans)

    apply (simp add: zle-zdiff-eq [symmetric])
    apply (insert 1)
    apply (rotate-tac −1)
    apply (drule int-less-le [THEN iffD1], simp)


    apply (simp add: zdiff-zle-eq)
    apply (cut-tac a = Rep-JavaInt a and b = Rep-JavaInt b
      in IntDiv.neg-mod-sign, simp add: 1)
    apply (subgoal-tac 0 ≤ MaxInt-int + Rep-JavaInt b)
    apply (blast intro: order-trans)
  apply (subgoal-tac MaxInt-int + MinInt-int + 1 ≤ MaxInt-int + Rep-JavaInt
```

```
b)

      apply (subgoal-tac 0 ≤ MaxInt-int + MinInt-int + 1)
      apply (blast intro: order-trans)
      apply (insert 2)
      apply (simp add: MaxInt-int-def MinInt-int-def BitLength-def)
      apply simp
      apply (simp add: add1-zle-eq)
      apply (cut-tac b = b in MinInt-leq)
      apply (rotate-tac −1)
      apply (drule zle-imp-zless-or-eq)
      apply auto
      done
    } note mod-minus-in-JavaInt2 = this

    moreover
    {
      assume 1: Rep-JavaInt b = 0
      have Rep-JavaInt a mod Rep-JavaInt b − Rep-JavaInt b ∈ JavaInt
        by (simp add: JavaInt-def mod-def DIVISION-BY-ZERO 1)
    } note mod-minus-in-JavaInt3 = this


    then show ?thesis
      apply (cases Rep-JavaInt b < 0)
      apply (simp add: mod-minus-in-JavaInt2)
      apply (simp add: HOL.linorder-not-less)
      apply (drule HOL.order-le-imp-less-or-eq)
      apply auto
      apply (simp add: mod-minus-in-JavaInt1)
      done
qed


lemma Rep-mod-homom:
  assumes 1: (0 < a & b < 0 | a < 0 & 0 < b)
  assumes 2: EX z. Rep-JavaInt a = z ∗ Rep-JavaInt b
  shows Rep-JavaInt (a mod b) = Rep-JavaInt a mod Rep-JavaInt b
  apply (insert 1 2)
  apply (simp add: mod-def split del: split-if)
  apply (simp add: Int-to-JavaInt-ident)
  done


lemma mod-Int-Abs-ident [simp]:
  Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) =
  Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)
```

by (*rule Int-to-JavaInt-ident*, *simp add*: *mod-in-JavaInt*)

lemma *div-Int-Abs-ident* :
assumes *A*: *Rep-JavaInt a* $\neq$ *MinInt-int* | *Rep-JavaInt b* $\neq -1$
shows *Int-to-JavaInt* (*Rep-JavaInt a div Rep-JavaInt b*) =
Abs-JavaInt (*Rep-JavaInt a div Rep-JavaInt b*)
apply (*rule Int-to-JavaInt-ident*)
apply (*rule div-in-JavaInt*, *simp*)
apply (*insert A*, *simp*)
done

15.3 Remainder Characterizations

lemma *JavaInt-mod-nondvd-pos-pos*:
assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* $\neq$ *0*
assumes *2*: *0* $<$ *a*
assumes *3*: *0* $<$ *b*
shows ((*a*::*JavaInt*) *mod b*)= *Abs-JavaInt* (*Rep-JavaInt a mod Rep-JavaInt b*)
proof −
show *?thesis*
apply (*insert 1 2 3*)
apply (*simp add*:*mod-def split del*: *split-if*)
apply (*subst if-not-P*)
apply (*simp-all add*: *Int-to-JavaInt-ident*)
apply (*rule disjI1*)
apply (*rule conjI*)
apply (*rule order-less-not-sym*)
defer *1*
apply (*rule order-less-not-sym*)
apply *simp-all*
done
qed

lemma *JavaInt-mod-nondvd-neg-neg*:
assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* $\neq$ *0*
assumes *2*: *a* $<$ *0*
assumes *3*: *b* $<$ *0*
shows ((*a*::*JavaInt*) *mod b*)= *Abs-JavaInt* (*Rep-JavaInt a mod Rep-JavaInt b*)
proof −
show *?thesis*
apply (*insert 1*,*insert 2*,*insert 3*)
apply (*simp add*:*mod-def split del*: *split-if*)
apply (*subst if-not-P*)
apply (*simp-all add*: *Int-to-JavaInt-ident*)
apply (*rule disjI1*)
apply (*rule conjI*)

```
  apply (rule order-less-not-sym)
  defer 1
  apply (rule order-less-not-sym)
  apply simp-all
  done
qed


lemma JavaInt-mod-nondvd-pos-neg:
  assumes 1: Rep-JavaInt a mod  Rep-JavaInt b ≠ 0
  assumes 2: 0 < a
  assumes 3: b < 0
  shows  ((a::JavaInt) mod b)= Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt
b) − b
proof −
show ?thesis
  apply (insert 1,insert 2,insert 3)
  apply (simp add:mod-def split del: split-if)
  apply (subst if-P)
  apply (simp-all add: Int-to-JavaInt-ident)

  apply (simp ! add: zmod-eq-0-iff zero-def)
  apply auto
  done
qed


lemma JavaInt-mod-nondvd-neg-pos:
  assumes 1: Rep-JavaInt a mod  Rep-JavaInt b ≠ 0
  assumes 2: a < 0
  assumes 3: 0 < b
  shows  ((a::JavaInt) mod b)= Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt
b) − b
proof −
show ?thesis
  apply (insert 1,insert 2,insert 3)
  apply (simp add:mod-def split del: split-if)
  apply (subst if-P)
  apply (simp-all add: Int-to-JavaInt-ident)

  apply (simp ! add: zmod-eq-0-iff zero-def)
  apply auto
  done
qed
```

16 Calculating Example Values for div and mod

16.1 Division and Modulo Lemmas: Base Cases

This lemma formalizes the property described in (2) for the division operator (JLS §15.17.2).

Again, the JLS is not very elaborate in (2) regarding the sign of the resulting value if the magnitude of the dividend is less than the magnitude of the divisor. It would have been clearer had they stated the result instead of letting the reader derive the result from the presented inequalities.

```
lemma MinInt-div-minusone :
  MinInt div −1 = MinInt
proof −
  have MinInt div −1 =
    Int-to-JavaInt (Rep-JavaInt MinInt div Rep-JavaInt (−1))
    apply (simp add: div-def)
    apply (simp add: number-of-def zero-def)
    apply (subst Int-to-JavaInt-ident [of −1])
    apply (simp add: minusone-in-JavaInt)
    apply (simp add: JavaInt-less-def JavaInt-le-def)
    apply (simp add: MinInt-def)
    apply auto
    apply(erule allE[of - −MinInt-int -])
    apply (thin-tac ?X)
    apply (thin-tac ?X)      — Isabelle/arith/bug
    apply simp
    done

  also have ... =
    Int-to-JavaInt
    (Rep-JavaInt MinInt div Rep-JavaInt (Int-to-JavaInt (−1)))
    by (simp add: number-of-def)

  also have ... =
    Int-to-JavaInt
    (Rep-JavaInt MinInt div
    Rep-JavaInt
    (Abs-JavaInt
    ((−1 + − MinInt-int) mod (2 ∗ − MinInt-int) + MinInt-int)))
    by (simp only: Int-to-JavaInt-def [of −1])

  also have ... = Int-to-JavaInt (Rep-JavaInt MinInt div (−1))
    apply (simp add: Rep-Abs-remove)
    apply (simp add: number-of-def MinInt-int-def MaxInt-int-def BitLength-def)
    done
  also have ... = Int-to-JavaInt (− Rep-JavaInt MinInt)
    by auto
  also have ... = MinInt
```

```
    apply (simp add: MinInt-def)

    done

  also from calculation
  show ?thesis by auto
qed


lemma JavaInt-div-dvd:
   assumes 1: Rep-JavaInt a mod  Rep-JavaInt b = 0
   shows      ((a::JavaInt) div b) = Int-to-JavaInt(Rep-JavaInt a div Rep-JavaInt
b)
proof -
show ?thesis
   apply (insert 1)
   apply (simp add:div-def split del: split-if)
   apply (subst if-not-P)
   apply (simp ! add: zmod-eq-0-iff zero-def,erule exE)
   apply (rule disjI2)
   apply auto
   done
qed


lemma JavaInt-0-div [simp]:
shows 0 div a = (0::JavaInt)
proof -
show ?thesis
   apply (subst JavaInt-div-dvd)
   apply (simp-all add: Int-to-JavaInt-ident zero-def)
   done
qed

lemma JavaInt-div-self [simp]:
assumes 1: a ≠ 0
shows      a div a = (1::JavaInt)
proof -
show ?thesis
   apply (subst JavaInt-div-dvd)
   apply (simp-all add: one-def 1)
   apply (subst zdiv-self)
   apply (simp-all add: Int-to-JavaInt-ident zero-def 1)
   apply (insert 1)
   apply (erule swap)
   apply (simp add: zero-def Rep-JavaInt-inject[symmetric] )
   done
qed
```

```
lemma JavaInt-div-by-zero [simp]:
shows a div 0 = (0::JavaInt)
proof -
show ?thesis
  apply (simp add:div-def zero-def DIVISION-BY-ZERO Int-to-JavaInt-ident
            split del: split-if)
  done
qed

lemma mod-zero1:
  assumes 1: ~ a < 0
  assumes 2: ~ 0 < a
  shows Rep-JavaInt a mod x = 0
  apply (insert 1 2)
  apply (drule neg-less-neg-greater-eq, simp)
  apply (simp add: zmod-zero zero-def)
  done

lemma mod-zero2:
  assumes 1: ~ a < 0
  assumes 2: ~ 0 < a
  shows x mod Rep-JavaInt a = x
  apply (insert 1 2)
  apply (drule neg-less-neg-greater-eq, simp)
  apply (simp add: DIVISION-BY-ZERO zero-def)
  done

lemma mod-zero3:
  assumes 1: ~ - a < 0
  assumes 2: ~ 0 < - a
  shows x mod Rep-JavaInt a = x
  apply (insert 1 2)
  apply (drule neg-less-neg-greater-eq, simp)
  apply (drule uminus-zero-zero, simp)
  apply (simp add: DIVISION-BY-ZERO zero-def)
  done

lemma mod-zero4:
  assumes 1: ~ a < 0
  assumes 2: ~ - a < 0
  shows x mod Rep-JavaInt a = x & x mod Rep-JavaInt (- a) = x
  apply (insert 1 2)
  apply (simp add: linorder-not-less)
  apply (drule neg-less-zero)
  apply (drule leq-geq-eq, simp)
  apply (drule uminus-zero-zero, simp)
  apply (simp add: DIVISION-BY-ZERO zero-def)
  done
```

lemma *mod-zero5*:
  assumes *1*: ~ *0* < *a*
  assumes *2*: ~ *0* < − *a*
  assumes *3*: *a* ≠ *MinInt*
  shows *x mod Rep-JavaInt a* = *x* & *x mod Rep-JavaInt* (− *a*) = *x*
  apply (*insert 1 2 3*)
  apply (*simp add*: *linorder-not-less*)
  apply (*drule pos-neg-eq*, *simp*+)
  apply (*simp add*: *DIVISION-BY-ZERO zero-def*)
  done

The following three properties show that JavaInt maintains the underlying properties *IntDiv.DIVISION-BY-ZERO*: *a div 0* = *0* ∧ *a mod 0* = *a*, *IntDiv.zmod-zero*: *0 mod ?b* = *0* and *IntDiv.zmod-self*: *?a mod ?a* = *0* of Integer.

lemma *DIVISION-BY-ZERO-JMOD* [*simp*]:
  shows *a mod* (*0*::*JavaInt*) = *a*
  by (*simp add*: *mod-def zero-def DIVISION-BY-ZERO Int-to-JavaInt-ident*
    Rep-JavaInt
    Rep-JavaInt-inverse)

lemma *mod-self*[*simp*]:
  shows (*a*::*JavaInt*) *mod a* = *0*
  by (*simp add*: *mod-def Int-to-JavaInt-ident zero-def*)

lemma *mod-zero*[*simp*]:
  shows *0 mod* (*a*::*JavaInt*) = *0*
  apply (*simp add*: *mod-def zero-def*)
  done

lemma *JavaInt-mod-0*:
   assumes *1*: *Rep-JavaInt a mod Rep-JavaInt b* = *0*
   shows ((*a*::*JavaInt*) *mod b*)= *0*
proof −
show *?thesis*
   apply (*insert 1*)
   apply (*simp add*:*mod-def split del*: *split-if*)
   apply (*subst if-not-P*)
   apply (*simp* ! *add*: *zmod-eq-0-iff zero-def*,*erule exE*)
   apply (*rule disjI2*)
   apply *auto*
   done
qed

In Java, the expression $c = MinInt\ mod\ a$ gives quite chaotic results. For the special case $a = 2^k$ for some k, it can be established that c is zero. For all other cases, we we have been unable to find any pattern.

lemma *mod-eq-0-iff2* :

```
  assumes 1: EX z. Rep-JavaInt x = z * Rep-JavaInt y
  shows     x mod y = 0
  apply (rule JavaInt-mod-0)
  apply (subst zmod-eq-0-iff)
  apply (simp add: 1 zmult-commute)
  done

lemma neg-charn1:
  assumes 1: neg (Rep-JavaInt n) = neg (Rep-JavaInt d)
  shows (Rep-JavaInt n < 0 & Rep-JavaInt d < 0) |
  (0 ≤ Rep-JavaInt n & 0 ≤ Rep-JavaInt d)
  apply (insert 1)
  apply (simp add: neg-eq-less-0)
  apply auto
  done

lemma neg-charn2:
 assumes 1: neg (Rep-JavaInt n) ≠ neg (Rep-JavaInt d)
  shows (Rep-JavaInt n < 0 & 0 ≤ Rep-JavaInt d) |
        (0 ≤ Rep-JavaInt n & Rep-JavaInt d < 0)
  apply (insert 1)
  apply (simp add: neg-eq-less-0)
  apply auto
  done


lemma Factor-in-JavaInt :
  assumes 1: Rep-JavaInt a = z * Rep-JavaInt b
  assumes 2: Rep-JavaInt b ≠ 0
  assumes 3: Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1
  shows     z ∈ JavaInt
  proof −
    {
      assume A: Rep-JavaInt a = MinInt-int
      assume A2: Rep-JavaInt b ≠ −1
      have    z ∈ JavaInt
        apply (insert 1 2)
     apply (drule divide-both-sides [of Rep-JavaInt a z * Rep-JavaInt b Rep-JavaInt
b])
        apply (drule zdiv-zmult-self1 [of Rep-JavaInt b z])
        apply simp
        apply (hypsubst)
        apply (rule div-in-JavaInt [of a b])
        apply (simp add: A A2)
        done
    } note factor-minint = this

    also
    {
```

```
    assume B1: Rep-JavaInt a = MinInt-int
    assume B2: Rep-JavaInt b = −1
    have z ∈ JavaInt
      apply (insert 3 B1 B2)
      apply simp
      done
    } note factor-minint-minusone = this

  moreover
  {
    assume B: Rep-JavaInt a ≠ MinInt-int
    assume C: Rep-JavaInt b ≠ 0
    have z ∈ JavaInt
      apply (insert 1 C)
    apply (drule divide-both-sides [of Rep-JavaInt a z ∗ Rep-JavaInt b Rep-JavaInt
b])
      apply (drule zdiv-zmult-self1 [of Rep-JavaInt b z])
      apply simp
      apply (hypsubst)
      apply (rule div-in-JavaInt [of a b])
      apply (simp add: B)
      done
  } note factor-nothing = this

  moreover
  {
    assume D: Rep-JavaInt b = 0
    have z ∈ JavaInt
      apply (insert 2 D)
      apply simp
      done
  } note factor-null = this

  show ?thesis
    apply (insert 1)
  apply (cases Rep-JavaInt a = MinInt-int)
  apply (cases Rep-JavaInt b = −1)
  apply (simp add: factor-minint-minusone)
  apply (simp-all add: factor-minint)
  — now: Rep-JavaInt a ≠ MinInt-int
  apply (cases Rep-JavaInt b ≠ 0)
  apply (simp-all add: factor-nothing)
  apply (simp-all add: factor-null)
  done
qed
```

These lemmas formalize the property described in (2) for the modulo operator (JLS §15.17.3).

lemma *MinInt-mod-minusone*[*simp*]:

```
  MinInt mod −1 = 0
    apply (simp add: mod-def)
    apply auto
    apply (simp add: JavaInt-less-def JavaInt-le-def number-of-def zero-def)
    apply (simp-all add: MinInt-def number-of-def zero-def)
    done
```

16.2 Sign of Divisions

```
lemma JavaInt-div-neg-shift:
   assumes 1:a ≠ MinInt
   assumes 2:b ≠ MinInt
   shows     a div (−b) = (−a) div b
proof −
{assume 1: a ≠ MinInt
 have      (Rep-JavaInt a mod  −(Rep-JavaInt b) = 0) =
           (Rep-JavaInt a mod  Rep-JavaInt b = 0)
   apply (cases Rep-JavaInt a mod Rep-JavaInt b = 0)
   apply (subst zmod-zminus2-eq-if)
   apply simp

   apply simp
   apply (subst zmod-zminus2-eq-if)
   apply simp

   apply (cases 0 < Rep-JavaInt b)
   apply (drule IntDiv.pos-mod-bound [of Rep-JavaInt b Rep-JavaInt a])
   apply simp

   apply (simp only: linorder-not-less)
   apply (drule HOL.order-le-imp-less-or-eq)
   apply (cases Rep-JavaInt b < 0)
   apply (drule IntDiv.neg-mod-bound [of Rep-JavaInt b Rep-JavaInt a])
   apply simp

   apply simp
   done
} note HL2 = this

{assume 1: a ≠ MinInt
 have      (−(Rep-JavaInt a) mod  Rep-JavaInt b = 0) =
           (Rep-JavaInt a mod  Rep-JavaInt b = 0)
   apply (insert 1)
   apply (drule HL2)
   apply (simp add: zmod-zminus2[of Rep-JavaInt a Rep-JavaInt b])
   done
} note HL3 = this

show ?thesis
```

```
  apply(cases Rep-JavaInt a mod  Rep-JavaInt b = 0)
  apply(simp add: 1 2 uminus-homom HL2 HL3 JavaInt-div-dvd zdiv-zminus2 )

  apply(cases a < 0)
  apply(cases b < 0)
  prefer 3
  apply(cases b < 0)

  apply(simp-all add: linorder-not-less order-le-less)
  apply auto

  apply(subst JavaInt-div-nondvd-neg-neg)
  apply(simp-all add: JavaInt-div-nondvd-neg-neg JavaInt-div-nondvd-neg-pos
                  JavaInt-div-nondvd-pos-neg JavaInt-div-nondvd-pos-pos
                  uminus-pos-if-neg uminus-neg-if-pos 1 2 uminus-homom
    HL2 HL3
                  zdiv-zminus2)
  done
qed
```

16.3 Sign of Remainders

The following two lemmas relate the signs of the modulo operands. Interestingly enough, they differ quite a lot from the corresponding lemmas on the *int* level, which are

IntDiv.zmod-zminus-zminus: $-$ *?a mod* $-$ *?b* $= -$ (*?a mod ?b*) and

IntDiv.zmod-zminus2: *?a mod* $-$ *?b* $= -$ ($-$ *?a mod ?b*)

```
lemma JavaInt-mod-neg1:
  assumes 1:a ≠ MinInt
  assumes 2:b ≠ MinInt
  shows   −((a::JavaInt) mod b) = (−a) mod b
proof −
{assume 1: a ≠ MinInt
 have     (−(Rep-JavaInt a) mod  Rep-JavaInt b = 0) =
          (Rep-JavaInt a mod  Rep-JavaInt b = 0)
  apply (simp add: zmod-zminus1-eq-if)
  apply (rule impI)
  apply (simp add: zdiff-eq-eq [of Rep-JavaInt b - 0])
  apply (cases 0 < Rep-JavaInt b)
  apply (drule IntDiv.pos-mod-bound [of Rep-JavaInt b Rep-JavaInt a])
  apply (simp add: HOL.neq-commute [of Rep-JavaInt b -]
    HOL.order-less-le)

  — 2nd case:
  apply (simp add: linorder-not-less)
  apply (cases Rep-JavaInt b = 0)
  apply (simp add: DIVISION-BY-ZERO)
```

— 3rd case:
apply (*drule conjI* [*of Rep-JavaInt b* $\leq$ *0 Rep-JavaInt b* $\neq$ *0*], *simp*)
apply (*simp add*: *HOL.order-less-le*[*symmetric*])
apply (*drule IntDiv.neg-mod-bound* [*of Rep-JavaInt b Rep-JavaInt a*])
apply *simp*
done
} **note** *HL2* = *this*
{have *Rep-JavaInt b* $\neq$ *MinInt-int*
apply (*insert 2*)
apply (*simp add*: *Rep-JavaInt-inject*[*symmetric*, *of b MinInt*])
apply (*simp add*: *MinInt-def Abs-JavaInt-inverse*)
done
} **note** *HL3* = *this*
{have *Rep-JavaInt a* $\neq$ *MinInt-int*
apply (*insert 1*)
apply (*simp add*: *Rep-JavaInt-inject*[*symmetric*, *of a MinInt*])
apply (*simp add*: *MinInt-def Abs-JavaInt-inverse*)
done
} **note** *HL3a* = *this*

{have *MinInt-int* + *Rep-JavaInt b* < *0*
apply (*simp only*: *zless-zdiff-eq*[*symmetric*,
of MinInt-int Rep-JavaInt b 0])
apply *simp*
done
} **note** *HL3b* = *this*

{have *Rep-JavaInt b* − *Rep-JavaInt a mod Rep-JavaInt b* $\in$ *JavaInt*
apply (*insert HL3*)
apply (*drule mod-minus-in-JavaInt*[*of b a*])
apply (*simp only*: *zminus-zdiff-eq*[*symmetric*, *of Rep-JavaInt b* -])
apply (*rule minus-in-JavaInt2* [*of*
(*Rep-JavaInt a mod Rep-JavaInt b*) − *Rep-JavaInt b*], *simp*)
apply (*simp add*: *zdiff-eq-eq*)
apply (*cases 0* < *Rep-JavaInt b*)
apply (*frule IntDiv.pos-mod-sign* [*of Rep-JavaInt b Rep-JavaInt a*])
apply (*insert MaxInt-MinInt-add*)
apply (*insert Rep-JavaInt* [*of b*])
apply (*drule MaxInt-int-is-largest* [*of Rep-JavaInt b*])
apply (*insert HL3b*)
apply (*drule order-less-le-trans* [*of MinInt-int* + *Rep-JavaInt b 0*
Rep-JavaInt a mod Rep-JavaInt b], *simp*)
apply *simp* — finished

— now prove the last subgoal:
apply (*simp add*: *linorder-not-less*)
apply (*cases Rep-JavaInt b* = *0*)
apply (*simp add*: *DIVISION-BY-ZERO*)
apply (*insert HL3a*, *simp*)

```
    apply (drule conjI[of Rep-JavaInt b ≤ 0 Rep-JavaInt b ≠ 0], simp)
    apply (simp add: order-less-le[symmetric])
    apply (drule IntDiv.neg-mod-bound [of Rep-JavaInt b Rep-JavaInt a])
    apply (insert MinInt-less0)
    apply (drule zless-not-sym [of MinInt-int 0])
    apply auto
 done
} note HL4 = this

show ?thesis
   apply(cases Rep-JavaInt a mod  Rep-JavaInt b = 0)
     apply(simp add: uminus-homom HL2 1 JavaInt-mod-0)
   — case Rep-JavaInt a mod Rep-JavaInt b ≠ 0
   apply(cases a < 0)
   apply(cases b < 0)
   prefer 3
   apply(cases b < 0)

   apply(simp-all add: linorder-not-less order-le-less)
   apply auto
   apply(simp-all add: JavaInt-mod-nondvd-neg-neg JavaInt-mod-nondvd-neg-pos
                       JavaInt-mod-nondvd-pos-neg JavaInt-mod-nondvd-pos-pos)

   apply (insert 1)
   apply (drule uminus-neg-if-pos) defer
   apply (drule uminus-neg-if-pos) defer
   apply (drule uminus-pos-if-neg, simp) defer
   apply (drule uminus-pos-if-neg, simp) defer
   apply (insert 1)
   apply (simp-all add: HL2[symmetric])
   apply (simp-all add: uminus-homom[symmetric])
   apply(simp-all add: JavaInt-mod-nondvd-neg-neg JavaInt-mod-nondvd-neg-pos
                       JavaInt-mod-nondvd-pos-neg JavaInt-mod-nondvd-pos-pos)
   apply (simp-all add: uminus-homom)
   apply(simp-all add: zmod-zminus1-eq-if)
   apply auto
   apply (simp-all only: JavaInt-add-commute [of − b -])
   apply (simp-all only: Ring.ring.minus-def[symmetric])
   apply (simp-all add: diff-def)
   apply (simp-all add: Int-to-JavaInt-ident HL4)
   apply (simp-all add: uminus-def)
   done
qed

lemma JavaInt-mod-neg2:
   shows   a mod (−b) = (a::JavaInt) mod b
proof −

  {
```

```
    fix x
    have (∀ y. (x::int) ≠ − y) ⟹ False
      by (drule spec [of - − x], auto)
  } note H1-1 = this

also
have H1 : EX q. Rep-JavaInt a = q ∗ Rep-JavaInt b ⟹
          EX q. Rep-JavaInt a = − q ∗ Rep-JavaInt b
  apply auto
  apply (simp only: zmult-zminus[symmetric])
  apply (simp only: zmult-cancel2)
  apply auto
  apply (drule H1-1)
  apply auto
  done

moreover
{
  fix z b
have ∀ q. − (z ∗ Rep-JavaInt b) ≠ q ∗ Rep-JavaInt b ⟹
  False
  by (drule spec [of - − z], auto)
} note H2 = this

{
  assume X1: b = MinInt
  have a mod (−b) = (a::JavaInt) mod b
    apply (insert X1)
    apply auto
    apply (simp add: uminus-MinInt)
    done
} note case-MinInt = this

{ fix a b
  assume X1: b ≠ MinInt
have Rep-JavaInt a mod Rep-JavaInt (− b) = Rep-JavaInt a mod − Rep-JavaInt
b
    apply (simp add: uminus-def)
    apply (subst Int-to-JavaInt-ident [of − Rep-JavaInt b])
    defer 1
    apply (subst Abs-JavaInt-inverse)
    defer 1
    apply simp
    apply (tactic distinct-subgoals-tac)
    apply (insert X1)
    apply (simp add: Rep-JavaInt-inject[symmetric] MinInt-def)
    done
} note reshape = this
```

```
{
 fix a b
 assume X1: ∀ z. Rep-JavaInt a ≠ z ∗ Rep-JavaInt b
 assume X2: b ≠ MinInt

 have
   Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− b)) =
       − b + Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)
   apply (subst reshape, simp add: X2)
   apply (simp add: add-def uminus-def)
   apply (subst Int-to-JavaInt-ident [of − Rep-JavaInt b])
   defer 1
   apply (subst Abs-JavaInt-inverse [of − Rep-JavaInt b])
   defer 1
   apply (subst zmod-zminus2-eq-if)
   apply (cases Rep-JavaInt a mod Rep-JavaInt b = 0)
   apply simp
   defer 1
   apply (subst if-not-P, simp)
    apply (simp only: zadd-commute [of − Rep-JavaInt b Rep-JavaInt a mod
Rep-JavaInt b]
     zdiff-def [symmetric])

   apply (tactic distinct-subgoals-tac)
   defer 1

   apply (insert X1)
   apply auto

   apply (insert X2)
   apply (simp add: Rep-JavaInt-inject[symmetric] MinInt-def)
   apply (subst Int-to-JavaInt-ident)
   apply auto
   apply (rule mod-minus-in-JavaInt)
   apply (simp add: not-MinInt-not-MinInt-int)
   done
} note L1 = this

{ fix a b q
 assume 1: Rep-JavaInt a = q ∗ Rep-JavaInt (− b)
 assume 2: b ≠ MinInt
 have (q ∗ Rep-JavaInt (− b)) mod Rep-JavaInt b = 0
   apply (simp add: uminus-def times-def)
   apply (insert 2)
   apply (subst Int-to-JavaInt-ident)
   defer 1
   apply (subst Abs-JavaInt-inverse)
   defer 1
   apply (simp only: mult-uminus-shift)
```

```
  apply (simp only: zmod-zmult-self1)
  apply (tactic distinct-subgoals-tac)
  apply (simp add: not-MinInt-not-MinInt-int)
  apply (simp add: minus-in-JavaInt)
  done
} note L2H1 = this

{
 assume X1: b ≠ MinInt
 assume X2: (0 < a ⟶ ~ b < 0) & (a < 0 ⟶ ~ 0 < b) |
     (EX z. Rep-JavaInt a = z * Rep-JavaInt b)
 assume X3: (0 < a ⟶ ~ - b < 0) & (a < 0 ⟶ ~ 0 < - b) |
     (EX z. Rep-JavaInt a = z * Rep-JavaInt (- b))

have  Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (- b)) =
      Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)

  apply (insert X2)
  apply (cases (0 < a ⟶ ~ b < 0) & (a < 0 ⟶ ~ 0 < b))
  apply (thin-tac ?X | ?Y)
  defer 1
  apply simp
  apply (drule Rep-mod-homom, simp)
  apply (drule mod-eq-0-iff2)

  apply (insert X3)
  apply (cases (0 < a ⟶ ~ - b < 0) & (a < 0 ⟶ ~ 0 < - b))
  apply (thin-tac ?X | ?Y)
  defer 1
  apply simp
  apply (drule Rep-mod-homom, simp)
  apply (drule mod-eq-0-iff2)
  apply simp

  apply auto

  apply (simp-all add: mod-zero1)
  apply (simp-all add: mod-zero2)
  apply (simp-all add: mod-zero3)
  apply (simp add: mod-zero5 X1)
  apply (simp add: mod-zero4)

  apply (subst L2H1 [of a - b], (simp add: X1)+ ) +

  apply (drule neg-less-neg-greater-eq [of b 0], simp)
  apply (simp add: uminus-def)
  apply (simp add: Int-to-JavaInt-ident minus-in-JavaInt zero-def)

  apply (subst reshape, simp add: X1)
```

```
    apply (simp add: zero-def)
    apply (simp add: zmod-zminus2-eq-if zero-def)

    apply (subst reshape, simp add: X1)
    apply (simp add: zero-def)
    apply (simp add: zmod-zminus2-eq-if zero-def)
    done
} note L2 = this

have stupid:
(Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) =
    b + Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− b)))
=
(Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− (− b))) =
    − (− b) + Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− b)))
  by (simp add: ring-simps)

{
  assume X1: b ≠ MinInt
  assume X2: ∀ z. Rep-JavaInt a ≠ z ∗ Rep-JavaInt (− b)

have
  b + Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− b)) =
        Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)
  apply (simp only: HOL.eq-commute [of
    b + Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt (− b)) -])
  apply (subst stupid)
  apply (rule L1)
  apply assumption
  apply (insert X1)
  apply (erule swap)
  apply simp
  apply (drule eq-times-eq [of − b MinInt −1])
  apply simp
  apply (simp only: MinInt-times-minusone)
  apply (simp only: uminus-times-minusone)
  apply (simp add: ring-simps)
  done
} note L3 = this

{
  assume X1: b ≠ MinInt
  assume X2: (0 < a & b < 0 | a < 0 & 0 < b)
  assume X3: (0 < a & − b < 0 | a < 0 & 0 < − b)

have
  False
  apply (insert X2 X3)
  apply (drule disjE [of - - False])
```

```
  apply simp-all
  apply (drule disjE [of - - False])
  prefer 4
  apply (drule disjE [of - - False])
  apply simp-all
  apply (drule conjE [of - - False], simp-all) +
  defer
  apply (drule conjE [of - - False], simp-all) +
  defer
  apply (drule conjE [of - - False], simp-all) +
  defer
  apply (drule conjE [of - - False], simp-all) +
  apply (insert X1)
  defer
  apply (drule uminus-neg1, simp)
  apply (drule less-greater-contradict, simp+)
  apply (drule uminus-neg2, simp)
  apply (drule less-greater-contradict, simp+)
  apply (drule uminus-neg1, simp)
  apply (drule less-greater-contradict, simp+)
  apply (drule less-greater-contradict, simp+)
  done
} note L4 = this

show ?thesis

  apply (cases b = MinInt)
  apply (rule case-MinInt, simp)

  apply (subst mod-def)
  apply (cases (0 < a & b < 0 | a < 0 & 0 < b)
    & ~ (EX z. Rep-JavaInt a = z * Rep-JavaInt b))
  apply simp — positive part is solved by simp
  defer 1
  apply (subst if-not-P, simp) — negative part needs some more help

  apply (simp)
  apply (subst mod-def)
  apply (cases (0 < a & - b < 0 | a < 0 & 0 < - b) &
          ~ (EX z. Rep-JavaInt a = z * Rep-JavaInt (- b)))
  apply simp — positive part is solved by simp
  defer 1
  apply (subst if-not-P, simp) — negative part needs some more help

  prefer 2
  apply (subst mod-def)
  apply (cases (0 < a & - b < 0 | a < 0 & 0 < - b) &
          ~ (EX z. Rep-JavaInt a = z * Rep-JavaInt (- b)))
  apply simp — positive part is solved by simp
```

```
  defer 1
  apply (subst if-not-P, simp) — negative part needs some more help
  apply simp
  apply (rule L1, simp)
  apply (simp,simp)

  apply (rule L2, simp)
  apply (simp,simp)

  apply (rule L3, simp)
  apply simp

  apply (drule L4)
  apply (simp, simp, simp)
  done
qed
```

16.4 Bounds of the Division Operation

```
lemma abs-explain:
  assumes A1: abs (Rep-JavaInt d) ≤ abs (Rep-JavaInt n)
  assumes A2: neg (Rep-JavaInt d) = neg (Rep-JavaInt n)
  shows (Rep-JavaInt d ≤ Rep-JavaInt n & 0 ≤ Rep-JavaInt n)
    | (Rep-JavaInt n ≤ Rep-JavaInt d & Rep-JavaInt n ≤ 0)
    apply (insert A1 A2)
    apply (drule neg-charn1)
    apply (simp add: zabs-def)
    apply auto
    done

lemma abs-explain-neq:
  assumes A1: abs (Rep-JavaInt d) ≤ abs (Rep-JavaInt n)
  assumes A2: neg (Rep-JavaInt d) ≠ neg (Rep-JavaInt n)
  shows (− Rep-JavaInt d ≤ Rep-JavaInt n & 0 ≤ Rep-JavaInt n)
    | (Rep-JavaInt n ≤ − Rep-JavaInt d & Rep-JavaInt n ≤ 0)
    apply (insert A1 A2)
    apply (drule neg-charn2)
    apply (simp add: zabs-def)
    apply auto
    done
```

This lemma formalizes the first property described in (1) for the division operator (JLS §15.17.2).

The predicate "neg" holds iff the value of its argument is less than zero.

Again, the phrasing in the JLS is quite imprecise as the "one special case that does not satisfy this rule" refers (in the text) to the lemma *quotient-sign-plus* as well as to the sister lemma *quotient-sign-minus* (see below). In fact, the lemma *quotient-sign-minus* is not touched by the special case $n = MinInt$

and $d = (-1::'a)$ because n and d have opposite signs. Therefore, we exclude the special case only in the lemma *quotient-sign-plus*.

In our total formalization, we also have to exclude the special case $d = (0::'a)$ because $a\ div\ (0::'a) = (0::'a)$ holds in our total world, which would contradict the lemma. Therefore, for $d = (0::'a)$ we cannot prove $(0::'a) < a\ div\ (0::'a)$. This does not really hurt as $d = (0::'a)$ is excluded in the exception layer anyways.

The JLS restricts the lemmas *quotient-sign-plus* and *quotient-sign-minus* to the cases where $|n| \geq |d|$ without specifying what happens for the other cases. Test runs showed that in Java, we always have $n\ div\ d = (0::'a)$ if $|d| > |n|$. This fact is missing in the JLS. Additionally, it could be checked here as well.

```
lemma quotient-sign-plus :
  assumes 1: (abs (Rep-JavaInt d) ≤ abs (Rep-JavaInt n))
  — we have to regard the int values, no wraparound!
  assumes 2: neg (Rep-JavaInt n) = neg (Rep-JavaInt d)
  assumes 3: n ≠ MinInt | d ≠ −1
  assumes 4: d ≠ 0
  shows 0 < (n div d)
proof −
  have HL4 [simp] : Rep-JavaInt n ≠ MinInt-int | Rep-JavaInt d ≠ (−1::int)
    apply (insert 3)
    apply (subst Abs-JavaInt-inject[symmetric, of Rep-JavaInt n MinInt-int] )
    apply (simp-all add: Rep-JavaInt)
    apply (subst Abs-JavaInt-inject[symmetric, of Rep-JavaInt d −1])
    apply (simp-all add: Rep-JavaInt)
    apply (simp add: Rep-JavaInt-inverse MinInt-def[symmetric] )
    done

  have HL3 [simp] : Rep-JavaInt n div Rep-JavaInt d ∈ JavaInt
    apply (rule div-in-JavaInt [of n d])
    apply (simp only: HL4)
    done

moreover {
  fix n d
  assume TL1-1: n = MinInt
  have (Rep-JavaInt n) ≤ (Rep-JavaInt d)
    apply (insert TL1-1 MinInt-is-least2[of d])
    apply (simp add: JavaInt-le-def)
    done
} note TL1 = this

{
  assume A1: 0 ≤ Rep-JavaInt n
  have 0 < Rep-JavaInt n
    apply (insert 1 A1 4)
```

```
  apply (simp add: Rep-JavaInt-inject[symmetric] zero-def)
  apply (drule abs-neq0-greater-neq0, assumption)
  apply (simp add: zabs-def)
  done
} note ineq2 = this

{
 assume A1: 0 ≤ Rep-JavaInt d
 have 0 < Rep-JavaInt d
  apply (insert 4 A1)
  apply (simp add: Rep-JavaInt-inject[symmetric] zero-def)
  done
} note ineq3 = this

{
 assume TL2-1: (Rep-JavaInt n < 0 & Rep-JavaInt d < 0) | (0 ≤ Rep-JavaInt
n & 0 ≤ Rep-JavaInt d)
 have n < 0 & d < 0 | 0 ≤ n & 0 ≤ d
  apply (insert TL2-1)
 apply (simp add: JavaInt-less-def JavaInt-le-def zero-def
  Rep-JavaInt-inject[symmetric])
 apply (simp only: int-less-le[of Rep-JavaInt d 0])
 apply (simp only: int-less-le[of Rep-JavaInt n 0])
 apply simp
 done
} note TL2 = this

{
 have n < 0 & d < 0 | 0 ≤ n & 0 ≤ d ⟶ ~(0 < n & d < 0 | n < 0 & 0 <
d)
  apply auto
  apply (simp add: linorder-not-le[symmetric, of n 0] order-less-imp-le)
  apply (simp add: linorder-not-le[symmetric, of d 0] order-less-imp-le)
  apply (simp add: linorder-not-le[symmetric, of d 0])
  apply (simp add: linorder-not-le[symmetric, of n 0])
  done
} note TL3 = this


{
 assume Absch2-1: Rep-JavaInt n < 0 & Rep-JavaInt d < 0
 have Rep-JavaInt n ≤ Rep-JavaInt d
 apply (insert 1 2)
 apply (drule abs-explain, simp)
 apply (insert Absch2-1)
 apply auto
 done
} note Absch2 = this
```

```
{ assume Absch3-1: Rep-JavaInt n < 0 ⟶ ~ Rep-JavaInt d < 0
  have 0 < Rep-JavaInt n
    apply (insert 2 Absch3-1)
    apply (drule neg-charn1)
    apply auto
    apply (simp-all add: linorder-not-less)
    apply (drule ineq2, simp)+
    done
} note Absch3 = this

{ assume Absch4-1: Rep-JavaInt n < 0 ⟶ ~ Rep-JavaInt d < 0
  have 0 < Rep-JavaInt d
    apply (insert 2 Absch4-1)
    apply (drule neg-charn1)
    apply auto
    apply (simp-all add: linorder-not-less)
    apply (drule ineq3, simp)+
    done
} note Absch4 = this

{
  assume Absch1-1: Rep-JavaInt n < 0 ⟶ ~ Rep-JavaInt d < 0
  have Rep-JavaInt d ≤ Rep-JavaInt n
    apply (insert Absch1-1)
    apply (frule Absch3)
    apply (drule Absch4)
    apply (insert 1)
  apply (drule abs-explain, simp add: 2)
  apply auto
  done
} note Absch1 = this


  show ?thesis

  apply (simp add: div-def split del: split-if)

  apply (insert 2)
  apply (drule neg-charn1)

  apply (drule TL2)

  apply (insert TL3)

  apply (drule mp [of n < 0 & d < 0 | 0 ≤ n & 0 ≤ d -], assumption)
  apply (simp add: if-not-P if-False)
  apply (thin-tac ?X)+
    — remove all assumptions
```

— remove Int-to-JavaInt-ident:
apply (*subst Int-to-JavaInt-ident*, *simp*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def*
Rep-JavaInt-inject[*symmetric*] *zero-def*)
apply (*simp only*: *int-less-le*[*symmetric*] *HOL.eq-commute*)

apply (*cases Rep-JavaInt n* $<$ *0* & *Rep-JavaInt d* $<$ *0*)
apply (*rule div-neg-neg-gt0*)
prefer *4*
apply (*rule div-pos-pos-gt0*)

apply *simp-all*

apply (*drule Absch1*, *simp*)
apply (*drule Absch3*, *simp*)
apply (*drule Absch4*, *simp*)
apply (*drule Absch2*, *simp*)
done

qed

lemma *null-less-MaxInt* [*simp*]: *0* $<$ *MaxInt-int*
by (*simp add*: *MaxInt-int-def BitLength-def*)

lemma *quotient-sign-minus1* :
assumes *1*: *Rep-JavaInt d* $<$ *0*
assumes *2*: *0* $<$ *Rep-JavaInt n*
assumes *3*: $-$ *Rep-JavaInt d* $\leq$ *Rep-JavaInt n*
shows (*n div d*) $<$ *0*

proof $-$

have *HL4* [*simp*] : *Rep-JavaInt n* $\neq$ *MinInt-int* | *Rep-JavaInt d* $\neq$ ($-$*1*::*int*)
apply (*insert 1 2*)
apply *auto*
apply (*insert MinInt-less0*)
apply *simp*
done

have *HL3* [*simp*] : *Rep-JavaInt n div Rep-JavaInt d* $\in$ *JavaInt*
apply (*rule div-in-JavaInt* [*of n d*])
apply (*simp only*: *HL4*)
done

```
  moreover{
    fix x y
    have ⟦ y ≤ x; x ≠ y ⟧ ⟹ y < (x::int)
      by (simp only: HOL.neq-commute[of x y])
  } note HL5 = this

  moreover
  have HL2 : ⟦ Rep-JavaInt n < 0; 0 ≤ Rep-JavaInt d ⟧
            ⟹ Rep-JavaInt n div Rep-JavaInt d + 1 ∈ JavaInt
   apply (simp add: JavaInt-def, auto)
   apply (insert HL4)
   apply (drule div-in-JavaInt [of n d])
   apply (simp add: JavaInt-def)
   apply auto

   apply (cases Rep-JavaInt d = 0)
   apply (simp !)
   apply (drule HL5[of 0 Rep-JavaInt d], simp)
   apply (drule div-neg-pos-less0 [of Rep-JavaInt n Rep-JavaInt d], simp)
   apply (simp add: add1-zle-eq)
   apply (insert null-leq-MaxInt)
   apply (drule order-less-le-trans [of - 0 MaxInt-int])
   apply simp-all
   done

moreover
 have HL2b : ⟦ 0 ≤ Rep-JavaInt n; Rep-JavaInt d < 0 ⟧
   ⟹ Rep-JavaInt n div Rep-JavaInt d + 1 ∈ JavaInt
   apply (simp add: JavaInt-def, auto)
   apply (insert HL4)
   apply (drule div-in-JavaInt [of n d])
   apply (simp add: JavaInt-def)
   apply (auto)
     — remove duplicate subgoals:
   apply (thin-tac [!] ?X ≠ ?Y)
   apply (tactic distinct-subgoals-tac)
     — prove the remaining subgoal:
   apply (drule div-nonneg-neg-le0, simp)
   apply (simp add: add1-zle-eq)
   apply (insert null-less-MaxInt)
   apply (drule order-le-less-trans [of - 0 MaxInt-int])
   apply simp-all
   done

moreover {
  fix x ::int
  have  EX z. − Rep-JavaInt d − Rep-JavaInt d ∗ x = z ∗ Rep-JavaInt d
    apply (rule exI [of - − 1 − x])
    apply (simp add: zdiff-zmult-distrib)
```

```
    apply (simp add: zmult-commute)
    done
} note HL1 = this

show ?thesis
   apply (simp add: div-def split del: split-if)
   apply (insert 1 2 3)
   — case distinction: rounding towards zero or not
   apply (cases (0 < n & d < 0 | n < 0 & 0 < d) &
              ~ (EX z. Rep-JavaInt n = z * Rep-JavaInt d))
   apply (simp-all add:if-not-P split del: split-if)
   apply (simp only: JavaInt-add-commute
     [of 1 Int-to-JavaInt (Rep-JavaInt n div Rep-JavaInt d)])
   apply (simp-all add: Int-to-JavaInt-ident Int-to-JavaInt-homom HL2b
                    JavaIntegersAdd.add-def zero-def one-def le-quasi-def)
   apply (simp-all add: neg-imp-zdiv-neg-iff, simp ! add:order-le-less)


   — remaining case: rounding.
   — Let − Rep-JavaInt d = Rep-JavaInt n
   apply (erule disjE, simp add: zless-iff-Suc-zadd[of − Rep-JavaInt d -])
   prefer 2
   — then dividable, no rounding should happen, contradiction
   apply(erule allE[of - −1 -])
   apply auto

  — Case: − Rep-JavaInt d < Rep-JavaInt n. Then Rep-JavaInt n = − Rep-JavaInt
d + (1 + int na).
   apply(subst zminus-zminus[symmetric, of Rep-JavaInt d])
   apply(subst zdiv-zminus2-eq-if,simp !  add: order-less-le)

   apply(subgoal-tac (− Rep-JavaInt d +(1+int na)) mod − Rep-JavaInt d ≠ 0)
   prefer 2
   apply (simp only: not-ex[symmetric],erule swap)
   apply (simp ! add: zmod-eq-0-iff,erule exE,simp add: HL1)

   apply (simp add: zdiff-zless-eq)
   apply (rule order-less-le-trans[of - 0 -], simp)
   apply (subst pos-imp-zdiv-nonneg-iff, simp,simp)

   done
qed


lemma quotient-sign-minus2 :
  assumes 1: 0 < Rep-JavaInt d
  assumes 2: Rep-JavaInt n < 0
  assumes 3: Rep-JavaInt n ≠ MinInt-int
  assumes 4: Rep-JavaInt n ≤ − Rep-JavaInt d
```

```
  shows      (n div d) < 0
proof −

have HL1: d ≠ MinInt
  apply (insert 1)
  apply (simp add: greater-zero-Abs-Rep[symmetric])
  apply (drule larger-zero-neq-MinInt [of d])
  apply (simp add: neq-commute)
done

show ?thesis
  apply (insert 3)
  apply (simp only: not-MinInt-not-MinInt-int[symmetric])
  apply(insert HL1)

  apply(subst uminus-uminus-ident[symmetric, of d])
  apply(subst  JavaInt-div-neg-shift)
  apply simp
  prefer 2
  apply(rule quotient-sign-minus1)

  apply(simp-all add: 1 2 3 4 uminus-homom)
  apply(subst zle-zminus)
  apply(rule 4)
  apply (simp add: neg-not-MinInt)
  done
qed
```

This lemma formalizes the second property described in (1) for the division operator (JLS §15.17.2).

Although the JLS does not explicitly mention this, we need not exclude the special case $n = MinInt$ and $d = (1::'a)$ here because we already assume that the signs of n and d are different.

```
lemma quotient-sign-minus :
  assumes 1: abs (Rep-JavaInt d) ≤ abs (Rep-JavaInt n)
  — we have to regard the values, no wraparound!
  assumes 2: neg (Rep-JavaInt n) ≠ neg (Rep-JavaInt d)
  assumes 3: n ≠ MinInt
  assumes 4: d ≠ 0
  shows (n div d) < 0

proof −

 {
   fix x
   assume h1: 0 ≤ (x::int)
   assume h2: neg x
   have False
```

```
    apply (insert h1 h2)
    apply (simp add: neg-eq-less-0)
    done
} note HL1 = this

{
  assume h1: 0 < Rep-JavaInt d
  have Rep-JavaInt n < 0
    apply (insert 2 h1)
    apply (simp add: neg-eq-less-0)
    done
} note HL2 = this

{
  assume h1: Rep-JavaInt d < 0
  have 0 < Rep-JavaInt n
    apply (insert 2 h1)
    apply (simp add: neg-eq-less-0)
    apply (simp add: linorder-not-less)
    apply (insert 1 4)
    apply (simp only: Rep-JavaInt-inject[symmetric])
    apply (simp only: zero-def Abs-JavaInt-inverse zero-in-JavaInt)
    apply (frule abs-neq0-greater-neq0, simp)
    apply (simp add: abs-def)
    done
} note HL3 = this

  show ?thesis

  apply (insert 1 2)
  apply (drule abs-explain-neq, simp)
  apply (insert 4)
  apply (simp add: Rep-JavaInt-inject[symmetric] zero-def)
  apply (simp add: zero-def[symmetric])

  apply auto
  apply (simp-all add: neg-eq-less-0)
  apply (erule swap, simp)
  apply (frule HL3)
  apply (rule quotient-sign-minus1, simp, simp, simp)

  apply (simp add: linorder-not-less)
  apply (simp only: HOL.neq-commute[of Rep-JavaInt d 0::int])
  apply (drule conjI[of 0 ≤ Rep-JavaInt d 0 ≠ Rep-JavaInt d], simp)
  apply (drule order-less-le[symmetric, THEN iffD1])
  apply (frule HL2)
  apply (rule quotient-sign-minus2, simp, simp, simp)
  defer
  apply (simp add: linorder-not-less [of Rep-JavaInt n 0])
```

```
  apply (drule zle-anti-sym[of Rep-JavaInt n 0], simp)
  apply (insert 1, simp)
  apply (simp only: zero-less-abs-iff[symmetric, of Rep-JavaInt d])
  apply (insert 3)
  apply (simp add: not-MinInt-not-MinInt-int)
  done
qed
```

16.5 Bounds on the Remainder

The following lemmas mirror the corresponding laws from IntDiv which are

$?b < 0 \Longrightarrow ?b < ?a \ mod \ ?b$
$?b < 0 \Longrightarrow ?a \ mod \ ?b \leq 0$
$0 < ?b \Longrightarrow ?a \ mod \ ?b < ?b$
$0 < ?b \Longrightarrow 0 \leq ?a \ mod \ ?b$

```
lemma neg-mod-bound:
  assumes 1: b < 0
  assumes 2: b ≠ MinInt
  shows     (b::JavaInt) < a mod b

proof -

{
  assume 1: Rep-JavaInt b < 0
  have 0 < - Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b
    apply (simp only: zadd-commute[of - Rep-JavaInt b -])
    apply (simp only: zdiff-zless-eq[symmetric, of 0 - - Rep-JavaInt b])
    apply (simp )
    apply (rule IntDiv.neg-mod-bound)
    apply (simp add: 1)
    done
} note HL1 = this


  have HL2: -1 - MaxInt-int = - (MaxInt-int + 1)
    by auto

{
  assume 1: Rep-JavaInt b < 0
  have - Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b ≤ MaxInt-int
    apply (simp only: zadd-commute[of - Rep-JavaInt b -])
    apply (simp only: zdiff-def[symmetric])
    apply (simp only: zdiff-zle-eq[of - Rep-JavaInt b MaxInt-int])
    apply (insert MinInt-leq [of b])
    apply (drule zadd-zle-mono1[of MinInt-int Rep-JavaInt b MaxInt-int])
    apply (simp add: JavaInt-maxmin-compare2)
    apply (insert 1)
    apply (drule IntDiv.neg-mod-sign[of Rep-JavaInt b Rep-JavaInt a])
```

```
  apply (cases −1 < Rep-JavaInt b + MaxInt-int)
  apply simp

  apply (simp add: zle-def [symmetric])
  apply (drule order.order-antisym[of −1 Rep-JavaInt b + MaxInt-int], simp)
  apply (simp add: zdiff-eq-eq [symmetric, of −1 Rep-JavaInt b MaxInt-int])
  apply (simp only: HL2)
  apply (simp add: JavaInt-maxmin-compare)
  apply (insert 2)
  apply (simp add: Rep-JavaInt-inject[symmetric,of b MinInt])
  apply (simp add: MinInt-def)
  done
} note HL3 = this

{
 assume H1: Rep-JavaInt b < 0
 have − Rep-JavaInt b + Rep-JavaInt a mod Rep-JavaInt b ∈ JavaInt
   apply (simp add: JavaInt-def)
   apply (rule conjI)
   apply (insert MinInt-leq0 H1)
   apply (drule HL1)
   apply (simp only: order-less-imp-le)
   apply (rule HL3, simp)
   done
} note HL4 = this

show ?thesis

— this way, we can cleanly distinguish the two cases of the mod definition:
  apply   (subst mod-def)
  apply (cases ((((0 < a) & (b < 0)) | ((a < 0) & (0 < b))) &
                (~ (EX z. ((Rep-JavaInt a) = (z ∗ (Rep-JavaInt b)))))))
  apply simp

  defer 1
  apply (subst if-not-P, simp)


  apply (simp-all only: Int-to-JavaInt-ident mod-in-JavaInt)
  apply (insert 1 2)
 apply (simp-all add: JavaInt-less-def JavaInt-le-def Rep-JavaInt-inject[symmetric]
 zero-def MinInt-def)
  apply (simp-all add: order-less-le[symmetric])

  apply (subst add-def)
  apply (simp add: Abs-JavaInt-inverse)
  apply (simp add: uminus-def)
  apply (simp add: Int-to-JavaInt-ident minus-in-JavaInt)
  apply (subst Int-to-JavaInt-ident)
```

```
  apply (rule HL4, simp)

  apply (subst Abs-JavaInt-inverse)
  apply (rule HL4, simp)

  apply (insert HL1)
  apply simp
  done
qed


lemma pos-mod-bound:
  assumes 1: 0 < b
  shows     (a::JavaInt) mod b < b
proof -

  have L1: (0::int) < Rep-JavaInt b ⟹
           Rep-JavaInt a mod Rep-JavaInt b − Rep-JavaInt b ∈ JavaInt
  apply (rule mod-minus-in-JavaInt [of b a])
  apply (simp add: MinInt-int-def BitLength-def)
  done

  have HL2: −1 − MaxInt-int = − (MaxInt-int + 1)
    by auto


have HL5: 0 < Rep-JavaInt b ⟹ − Rep-JavaInt b ∈ JavaInt
  apply (rule minus-in-JavaInt)
  apply (insert MinInt-less0)
  apply (drule order-less-trans[of MinInt-int 0 Rep-JavaInt b], simp)
  apply (simp)
  done

    show ?thesis
  apply   (insert 1)
  apply   (simp add: mod-def le-quasi-def)
  apply   (rule conjI)
  apply   (rule impI)
  prefer  2
  apply   (rule impI)
  apply   (thin-tac ?X ⟶ ?Y)
  prefer  2
  apply   (thin-tac ?X & ?Y)
  defer   1
  apply (rule IntDiv.pos-mod-bound)
  apply (simp-all add: zero-def)
  apply (subst add-def)
  apply (simp add: Abs-JavaInt-inverse)
```

```
  apply (simp add: uminus-def)
  apply (simp add: Int-to-JavaInt-ident [of − Rep-JavaInt b] HL5)
  apply (subst Int-to-JavaInt-ident)
  apply (rule mod-minus-divisor-in-JavaInt, simp)
  apply (subst Abs-JavaInt-inverse)
  apply (rule mod-minus-divisor-in-JavaInt, simp)

  apply  auto
  apply (subgoal-tac Rep-JavaInt b < 2 ∗ Rep-JavaInt b)
  apply (drule IntDiv.pos-mod-bound [of Rep-JavaInt b Rep-JavaInt a])
  apply (blast intro: zless-trans)
  apply auto
  done
qed
```

The phrases stated in (3) in the JLS description of the modulo operator (JLS §15.17.3) are not at all clear to us. We formalized them as follows in neg-mod-sign and pos-mod-sign, in the hope of meeting the intentions of the authors:

```
lemma neg-mod-sign:
  assumes 1: a < 0
  assumes 2: b ≠ 0
  shows     (a::JavaInt) mod b ≤ 0
proof −

  {
    assume 1: a < 0
    assume 2: b < 0
    have     (a::JavaInt) mod b ≤ 0
      apply (insert 1 2)
      apply (simp add: mod-def)
      apply (auto)
      prefer 3
      apply (simp-all add: zero-def JavaInt-le-def le-quasi-def)
      done
  } note neg-mod-sign1 = this


  {
    assume 1: 0 < b
    assume 2: a < 0
    have     (a::JavaInt) mod b ≤ 0
      apply (insert 1 2)
      apply (cases EX z. Rep-JavaInt a = z ∗ (Rep-JavaInt b))
      apply (subst mod-eq-0-iff2, simp)
      apply simp
      apply (simp add: mod-def)
      apply (simp add: uminus-def zero-def JavaInt-le-def le-quasi-def)
      apply (subst Int-to-JavaInt-ident, simp)
```

```
    apply (simp only: add-def)
    apply (subst Abs-JavaInt-inverse, simp) +
    apply (simp only: zadd-commute)
    apply (subst Int-to-JavaInt-ident, simp)
    apply (subst Abs-JavaInt-inverse, simp)

     apply (simp only: zadd-commute [of − Rep-JavaInt b Rep-JavaInt a mod
Rep-JavaInt b]
    zdiff-def[symmetric])
    apply (subst zle-iff-zdiff-zle-0[symmetric])
    apply (subst int-le-less)
    apply (rule disjI1)
    apply (rule IntDiv.pos-mod-bound)
    apply simp
    done
 } note neg-mod-sign2 = this

 show ?thesis
   apply   (insert 1 2)
   apply   (cases b < 0)
   prefer 2
   apply   (drule linorder-not-less [THEN iffD1])
   apply   (drule order-le-less [THEN iffD1])
   apply   (auto simp: neg-mod-sign1 neg-mod-sign2)
   done
qed


lemma pos-mod-sign:
  assumes 1: 0 ≤ a
  assumes 2: b ≠ 0
  assumes 3: b ≠ MinInt
  shows     0 ≤ (a::JavaInt) mod b
proof −
  {
    assume 1: 0 ≤ a
    assume 2: 0 < b
    have     0 ≤ (a::JavaInt) mod b
      apply (insert 1 2)
      apply (cases a=0)
      apply simp
      apply (simp add: mod-def)
      apply (auto)
      apply (drule order-less-imp-not-less [of 0 b], simp)

      apply (drule order-less-imp-not-less [of a 0])
      apply (drule order-le-less [of 0 a, THEN iffD1])
      apply (simp only: HOL.neq-commute)
```

```
    apply simp

    apply (simp only: linorder-not-less[of 0 a])
    apply (drule order-antisym [of 0 a], simp)
    apply (simp add: HOL.neq-commute)

    apply (simp only: zero-def JavaInt-le-def le-quasi-def)
    apply (subst Abs-JavaInt-inverse[of Rep-JavaInt a mod Rep-JavaInt b])
    apply (simp only: mod-in-JavaInt)
    apply (simp only: Abs-JavaInt-inverse[of 0] zero-in-JavaInt) +
    apply (rule IntDiv.pos-mod-sign)
    apply (assumption)
    apply (simp add: zero-def)
    done
  } note pos-mod-sign1 = this

{
  assume 1: 0 ≤ a
  assume 2: b < 0
  assume 3: b ≠ MinInt
  have     0 ≤ (a::JavaInt) mod b
  apply (insert 1 2 3)
  apply (cases a = 0)
  apply simp
  apply (drule order-le-imp-less-or-eq)
  apply simp
  apply (cases EX z. Rep-JavaInt a = z ∗ (Rep-JavaInt b))
  apply (subst mod-eq-0-iff2)
  apply auto
  apply (simp add: mod-def)

  apply (simp-all add: uminus-def zero-def le-quasi-def MinInt-def)
  apply (simp add: Rep-JavaInt-inject[symmetric])
  apply (subst Int-to-JavaInt-ident)
  apply (rule minus-in-JavaInt, simp)
  apply (simp add: add-def)
  apply (subst Abs-JavaInt-inverse)
  apply (rule minus-in-JavaInt, simp)
  apply (subst Int-to-JavaInt-ident)
  apply simp
  apply (rule mod-minus-in-JavaInt, simp)

  apply (simp add: JavaInt-le-def)
  apply (subst Abs-JavaInt-inverse)
  apply (rule mod-minus-in-JavaInt, simp)
  apply (simp add:zle-zdiff-eq)
  apply (subst int-le-less)
  apply (rule disjI1)
  apply (rule IntDiv.neg-mod-bound)
```

```
  apply assumption
  done
} note pos-mod-sign2 = this


  show ?thesis
    apply  (insert 1 2 3)
    apply  (cases b < 0)
    prefer 2
    apply  (drule linorder-not-less [THEN iffD1])
    apply  (tactic rotate-tac ~1 1)
    apply  (drule order-le-less [THEN iffD1])
    apply  (auto simp: pos-mod-sign1 pos-mod-sign2)
    done
qed
```

This lemma formalizes the property described in (4) for the modulo operator (JLS §15.17.3).

It is not clear whether the "magnitude of the result" refers to the mathematical absolute value or to the Java method `Math.abs`. We decided to use the function *abs* on JavaInt which allows us to stay in the abstract model. This has the drawback that the lemma cannot be used for b = MinInt because $|MinInt| = MinInt$, therefore there exists no JavaInt value which is strictly smaller than $|MinInt|$.

```
lemma JavaInt-mod-less:
  assumes 1: b ≠ 0
  assumes 2: b ≠ MinInt
  shows     abs ((a::JavaInt) mod b) < abs b
proof −

{
 assume L5-1: (0 ≤ a)
 assume L5-2: (b < 0)
 have a mod b < (− b)
   apply (insert 1 2 L5-1 L5-2)
  apply (subst JavaInt-mod-neg2[symmetric] )
  apply (rule pos-mod-bound)
  apply (rule uminus-pos-if-neg, simp+ )
  done
} note L5 = this

{
 assume L6-2: (0 < b)
 have a mod b < b
   apply (insert L6-2)
   apply (rule pos-mod-bound, simp)
   done
} note L6 = this
```

```
{
fix a b
assume L3-1: (a < (0::JavaInt))
assume L3-2: (b < (0::JavaInt))
assume L3-3: (b ≠ 0)
assume L3-4: (b ≠ MinInt)

have − (a mod b) < (− b)
  apply (insert L3-1 L3-2 L3-3 L3-4)
  apply (insert MinInt-is-least2[of b])
   apply (tactic rotate-tac ~1 1)
   apply (drule order-le-less [THEN iffD1])
   apply auto
   apply (subst swap-le)
  apply (insert MinInt-is-least2[of b])
   apply (simp add: less-vs-eq-or-le less-vs-not-le[symmetric])
   apply (auto)
   prefer 2
   apply (rule neg-mod-bound)
   apply auto
   apply (drule neg-mod-bound [of b a], simp)
   apply auto
   done
} note L3 = this

{
assume L4-1: (a < 0)
assume L4-2: (0 < b)
have − (a mod b) < b
  apply (subst JavaInt-mod-neg2[symmetric])
  apply (subst uminus-uminus-ident[symmetric, of b])
  apply (subst swap-le[of a mod (− (− (− b))) − b])
  apply (simp-all)
  — discharge 1st condition:
  apply (insert L4-2 2)
  apply (drule uminus-neg-if-pos[of b])
  apply (simp add: neg-not-MinInt[symmetric,of b])
  apply (drule neg-mod-bound [of − b a], simp)
  apply auto
  — discharge 2nd condition:
  apply (simp add: neg-not-MinInt[symmetric,of b])
  — discharge rest:
  apply (drule uminus-neg-if-pos[of b])
  apply (simp add: neg-not-MinInt[symmetric,of b])
  apply (drule neg-mod-bound [of − b a], simp+ )
  done
} note L4 = this
```

```
{
assume L1-1: 0 ≤ a
have abs (a mod b) < abs b
  apply (insert 1 2 L1-1)
   apply      (subst abs-pos [of a mod b])
   apply      (auto intro: pos-mod-sign)
   apply      (cases 0 ≤ b)
   apply      (auto simp: abs-neg linorder-not-le)
   apply (subst abs-pos [of b], simp)
   apply      (simp add:  order-le-less [of 0 b])
   prefer     2
   apply      (subst abs-neg)
   apply (simp add: order-less-imp-le)
   apply (drule L5, simp+ )
   apply (drule L6, simp+ )
   done
} note L1 = this


{
assume L2-1: ~ (0 ≤ a)
have abs (a mod b) < abs b
  apply (insert 1 2 L2-1)
   apply      (drule linorder-not-le [THEN iffD1])
   — now resolve abs'es
   apply      (subst abs-neg [of a mod b])
   apply      (auto intro: neg-mod-sign)
   apply      (cases 0 ≤ b)
   apply      (auto simp: abs-pos linorder-not-le)
   apply      (simp add:  order-le-less)
   prefer     2
   apply      (subst abs-neg)
   apply (simp add: order-less-imp-le)
   apply (drule L3, simp+ )
   apply (drule L4, simp+ )
   done
} note L2 = this
show ?thesis
   apply      (insert 1 2)
   apply      (cases 0 ≤ a)
   apply (drule L1, simp)
   apply (drule L2, simp)
   done
qed
```

16.6 Main Theorem: The div/mod lemma

```
lemma MinInt-minusone-div-mod-eq :
  (MinInt div −1) ∗ (−1) + (MinInt mod −1) = MinInt
```

```
  by (simp add: MinInt-mod-minusone MinInt-div-minusone
    MinInt-times-minusone)

lemma eq-znull-eq-jnull:
(Rep-JavaInt a = (0::int)) = (a = (0::JavaInt))
  by (simp add: Rep-JavaInt-inject [symmetric] Abs-JavaInt-inverse zero-def)
```

This lemma formalizes the property described in (1) for the modulo operator (JLS §15.17.3).

```
lemma JavaInt-div-mod-result :
  ((a::JavaInt) div b) * b + (a mod b) = a
proof -

  have * : ((((Int-to-JavaInt ((Rep-JavaInt a) div (Rep-JavaInt b))) * b) +
    b) +
    ((Int-to-JavaInt ((Rep-JavaInt a) mod (Rep-JavaInt b))) - b))
    = ((((Int-to-JavaInt ((Rep-JavaInt a) div (Rep-JavaInt b))) * b)) +
    ((Int-to-JavaInt ((Rep-JavaInt a) mod (Rep-JavaInt b)))))
  proof -
    have ((((Int-to-JavaInt ((Rep-JavaInt a) div (Rep-JavaInt b))) * b) +
      b) +
      ((Int-to-JavaInt ((Rep-JavaInt a) mod (Rep-JavaInt b))) + - b)) =
      ((((Int-to-JavaInt ((Rep-JavaInt a) div (Rep-JavaInt b))) * b) +
      b) +
      ((- b) + (Int-to-JavaInt ((Rep-JavaInt a) mod (Rep-JavaInt b)))))
      (is - = (?x + b) + ((- b) + ?y))
      by (simp only: JavaInt-add-commute)
    also have ... =  ?x + (b + ((- b) + ?y))
      by (simp add: JavaInt-add-assoc)
    also have ... =  ?x + (b + (- b) + ?y)
      by (simp add: JavaInt-add-assoc [symmetric])
    also have ... = ?x + ?y
      by simp
    also from calculation show ?thesis by auto
  qed

  moreover
  { fix a b
    have
      Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a div Rep-JavaInt b * Rep-JavaInt b)) +
      Rep-JavaInt (Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b))))
      = Rep-JavaInt a
    proof -
      have
        Rep-JavaInt (Int-to-JavaInt
        (Rep-JavaInt (Int-to-JavaInt
        (Rep-JavaInt a div Rep-JavaInt b * Rep-JavaInt b)) +
```

Rep-JavaInt (*Int-to-JavaInt* (*Rep-JavaInt a mod Rep-JavaInt b*)))) =
((*Rep-JavaInt a div Rep-JavaInt b* ∗ *Rep-JavaInt b* − *MinInt-int*)
mod − ((*2*::*int*) ∗ *MinInt-int*) + *MinInt-int* +
((*Rep-JavaInt a mod Rep-JavaInt b* − *MinInt-int*)
mod − ((*2*::*int*) ∗ *MinInt-int*) + *MinInt-int*) −
MinInt-int) *mod* − ((*2*::*int*) ∗ *MinInt-int*) + *MinInt-int*
(**is** *Rep-JavaInt* (*Int-to-JavaInt* (*Rep-JavaInt* (*Int-to-JavaInt ?rdiv*)
+ *Rep-JavaInt* (*Int-to-JavaInt ?rmod*))) =
((*?rdiv* − *?mi*) *mod ?zmi* + *?mi* +
((*?rmod* − *?mi*) *mod ?zmi* + *?mi*)
− *?mi*) *mod ?zmi* + *?mi*)
apply (*simp add*: *Int-to-JavaInt-def*)
apply (*subst Rep-Abs-remove*)+
apply *auto*
done
moreover have
... = ((*?rdiv* − *?mi*) *mod ?zmi* +
((*?rmod* − *?mi*) *mod ?zmi* + *?mi* + *?mi*)
− *?mi*) *mod ?zmi* + *?mi*
by (*simp add*: *zadd-commute*)
moreover have
... = ((*?rdiv* − *?mi*) *mod ?zmi* +
(*?rmod* − *?mi*) *mod ?zmi* + *?mi* + *?mi*
− *?mi*) *mod ?zmi* + *?mi*
by (*simp add*: *zadd-assoc*)
moreover have
... = ((*?rdiv* − *?mi*) *mod ?zmi* +
(*?rmod* − *?mi*) *mod ?zmi* + *?mi*
) *mod ?zmi* + *?mi*
by (*simp add*: *zadd-commute*)
moreover have
... = (((*?rdiv* − *?mi*) *mod ?zmi* +
(*?rmod* − *?mi*) *mod ?zmi*)
+ *?mi*
) *mod ?zmi* + *?mi*
(**is** - = (*?rdivmod* + *?mi*) *mod ?zmi* + *?mi*)
by *auto*
moreover have
... = ((*?rdivmod mod ?zmi*)
+ *?mi*
) *mod ?zmi* + *?mi*
by (*subst zmod-zadd-left-eq*[*symmetric*], *auto*)
moreover have ... = (((*?rdiv* − *?mi* + *?rmod* − *?mi*) *mod ?zmi*)
+ *?mi*
) *mod ?zmi* + *?mi*
by (*simp add*: *zmod-zadd1-eq* [*symmetric*])
moreover have ... = (((*Rep-JavaInt b* ∗ (*Rep-JavaInt a div Rep-JavaInt b*)

+ *?rmod* − *?mi* − *?mi*) *mod ?zmi*)

```
      + ?mi
      ) mod ?zmi + ?mi
      by (simp add: zadd-commute zmult-commute)
    moreover have ... = (  ((Rep-JavaInt a − ?mi − ?mi) mod ?zmi)
      + ?mi
      ) mod ?zmi + ?mi
      by (simp add: zmod-zdiv-equality [symmetric])
    moreover have ... = (  Rep-JavaInt a − ?mi − ?mi
      + ?mi
      ) mod ?zmi + ?mi
      by (simp del: Minustwo-Minus-two add: zmod-zadd-left-eq [symmetric])
    moreover have ... = ( Rep-JavaInt a − ?mi ) mod ?zmi + ?mi
      by (simp add: zadd-commute zadd-zminus-inverse2)
    moreover have ... = Rep-JavaInt a
      apply (subst Rep-Abs-remove [symmetric])
      apply (simp-all only: Rep-JavaInt-inject
        zdiff-def zmult-zminus-right [symmetric]
        Int-to-JavaInt-def [THEN meta-eq-to-obj-eq, symmetric]
        Int-to-JavaInt-ident Rep-JavaInt Rep-JavaInt-inverse)
      done
    ultimately show ?thesis by auto
  qed
} note proof-core = this

moreover — proof for special case
{ fix a b
  have Rep-JavaInt b = 0 ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
    apply (simp add: IntDiv.DIVISION-BY-ZERO)
    apply (simp add: Rep-JavaInt)
    done
} note zero-proof = this

moreover  — main proof of one of the four subgoals
{ fix a b
  have
    ⟦ 0 ≤ Rep-JavaInt a; Rep-JavaInt b < 0 ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1:  0 ≤ Rep-JavaInt a
    assume 2:  Rep-JavaInt b < 0

    have fold1: Rep-JavaInt
```

```
      (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) * b +
      Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
      Rep-JavaInt
      (Int-to-JavaInt
      (Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a div Rep-JavaInt b * Rep-JavaInt b)) +
      Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a mod Rep-JavaInt b))))
      apply (insert 1 2)
      apply (simp add: times-def add-def)
      apply (simp add: Int-to-JavaInt-ident
        div-Int-Abs-ident [of a b]
        MinInt-int-def BitLength-def)
      apply (simp add: Abs-JavaInt-inverse
        div-in-JavaInt [of a b]
        MinInt-int-def BitLength-def)
      done

    then show ?thesis
      by (simp only: fold1 mod-Int-Abs-ident [symmetric] proof-core)
  qed
} note pos-neg-proof = this

moreover  — main proof of one of the four subgoals
{ fix a b
  have
    ⟦ 0 ≤ Rep-JavaInt a; 0 < Rep-JavaInt b ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) * b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: 0 ≤ Rep-JavaInt a
    assume 2: 0 < Rep-JavaInt b

    have fold1: Rep-JavaInt
      (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) * b +
      Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
      Rep-JavaInt
      (Int-to-JavaInt
      (Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a div Rep-JavaInt b * Rep-JavaInt b)) +
      Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a mod Rep-JavaInt b))))
      apply (insert 1 2)
      apply (simp add: times-def add-def)
      apply (simp add: Int-to-JavaInt-ident
        div-Int-Abs-ident [of a b]
        MinInt-int-def BitLength-def)
```

```
      apply (simp add: Abs-JavaInt-inverse
        div-in-JavaInt [of a b]
        MinInt-int-def BitLength-def)
      done

    then show ?thesis
      apply (insert 1 2)
      apply (simp only: fold1 mod-Int-Abs-ident [symmetric] proof-core)
      done
  qed
} note nonneg-pos-proof = this

moreover  — main proof of one of the four subgoals
{ fix a b
  have
    ⟦ Rep-JavaInt a < 0; 0 < Rep-JavaInt b ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: Rep-JavaInt a < 0
    assume 2: 0 < Rep-JavaInt b

    have fold1: Rep-JavaInt
      (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
      Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
      Rep-JavaInt
      (Int-to-JavaInt
      (Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a div Rep-JavaInt b ∗ Rep-JavaInt b)) +
      Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a mod Rep-JavaInt b))))
      apply (insert 1 2)
      apply (simp add: times-def add-def)
      apply (simp add: Int-to-JavaInt-ident
        div-Int-Abs-ident [of a b]
        MinInt-int-def BitLength-def)
      apply (simp add: Abs-JavaInt-inverse
        div-in-JavaInt [of a b]
        MinInt-int-def BitLength-def)
      done

    then show ?thesis
      by (simp only: fold1 mod-Int-Abs-ident [symmetric] proof-core)
  qed
} note neg-pos-proof = this

moreover — groups nonneg-pos-proof and neg-pos-proof
```

```
{ fix a b
  have
    ⟦ 0 < Rep-JavaInt b ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: 0 < Rep-JavaInt b
    show ?thesis
      apply (insert 1)
      apply (cases 0 ≤ Rep-JavaInt a)
      apply (simp only: mod-Int-Abs-ident)
      apply (rule nonneg-pos-proof, auto)
      apply (simp only: JavaInt-times-commute [of b -])
      apply (subst JavaInt-add-commute)
      apply (rule neg-pos-proof)
      apply auto
    done
  qed
} note all-pos-proof = this

moreover — main proof of one of the four subgoals
{ fix a b
  have
    ⟦ Rep-JavaInt a < 0; Rep-JavaInt b < 0; Rep-JavaInt a ≠ MinInt-int |
Rep-JavaInt b ≠ −1 ⟧
    ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: Rep-JavaInt a < 0
    assume 2: Rep-JavaInt b < 0
    assume 3: Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1

    have fold1: Rep-JavaInt
      (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
      Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
      Rep-JavaInt
      (Int-to-JavaInt
      (Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a div Rep-JavaInt b ∗ Rep-JavaInt b)) +
      Rep-JavaInt (Int-to-JavaInt
      (Rep-JavaInt a mod Rep-JavaInt b))))
      apply (insert 1 2 3)
      apply (simp add: times-def add-def)
      apply (simp add: Int-to-JavaInt-ident
```

```
        div-Int-Abs-ident [of a b]
        MinInt-int-def BitLength-def)
      apply (simp add: Abs-JavaInt-inverse
        div-in-JavaInt [of a b]
        MinInt-int-def BitLength-def)
      done

    then show ?thesis
      by (simp only: fold1 mod-Int-Abs-ident [symmetric] proof-core)
  qed
} note neg-neg-proof = this

moreover  — proof of special case MinInt div −1
have MinInt-minusone-proof:
  ⟦ Rep-JavaInt a = MinInt-int; Rep-JavaInt b = −1 ⟧ ⟹
  Rep-JavaInt
  (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
  Int-to-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
  Rep-JavaInt a
proof −
  assume 1: Rep-JavaInt a = MinInt-int
  assume 2: Rep-JavaInt b = −1
  show ?thesis
    apply (simp add: 1 2 times-def Int-to-JavaInt-def add-def)
    apply (simp add: Abs-JavaInt-inverse
      JavaInt-def MaxInt-int-def MinInt-int-def
      BitLength-def) +
    done
qed


moreover
{ fix a b
  have
    ⟦ Rep-JavaInt a ≤ 0 ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
     Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: Rep-JavaInt a ≤ 0
    show ?thesis
      apply (cases Rep-JavaInt a < 0)
      apply (cases Rep-JavaInt b < 0)
      apply (cases Rep-JavaInt a ≠ MinInt-int)
      apply (rule neg-neg-proof, auto)
      apply (cases Rep-JavaInt b = −1)
      — Ra = MinInt, Rb = (−1::'a)
      apply (simp add: MinInt-minusone-proof)
```

```
    apply (simp add: times-def)
    apply (simp add: Int-to-JavaInt-def)
    apply (simp add: MinInt-def)
    apply (simp add: Minustwo-Minus-two zero-def[symmetric]) +
    — Ra = MinInt, Rb ≠ (−1::'a)
    apply (tactic res-inst-tac [(t,MinInt-int)] subst 1)
    apply (tactic atac 1)
    apply (subst JavaInt-times-commute)
    apply (subst JavaInt-add-commute)
    apply (rule neg-neg-proof)
    apply simp-all

    — Ra < (0::'a); (0::'a) ≤ Rb
    apply (cases Rep-JavaInt b = 0)
    apply (subst JavaInt-times-commute [of b -])
    apply (subst JavaInt-add-commute)
    apply (rule zero-proof, auto)

    apply (subst JavaInt-times-commute [of b -])
    apply (subst JavaInt-add-commute)
    apply (rule neg-pos-proof, auto)
    apply (insert 1)
    apply (simp add: zle-def [symmetric])
    apply (drule zle-anti-sym, simp)
    apply auto
    done
  qed
} note nonpos-all-proof = this

moreover
{ fix a b
  have
    ⟦ 0 ≤ Rep-JavaInt a ⟧ ⟹
    Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) ∗ b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
  proof −
    assume 1: 0 ≤ Rep-JavaInt a
    show ?thesis
      apply (insert 1)
      apply (cases 0 < Rep-JavaInt b)
      apply (rule nonneg-pos-proof, auto)
      apply (cases Rep-JavaInt b = 0)
      apply (subst JavaInt-times-commute [of b -])
      apply (subst JavaInt-add-commute)
      apply (rule zero-proof, auto)
      apply (rotate-tac)
      apply (simp add: zle-def [symmetric])
```

```
    apply (drule zle-imp-zless-or-eq, simp)
    apply (subst JavaInt-times-commute [of b -])
    apply (subst JavaInt-add-commute)

    apply (rule pos-neg-proof, auto)

    done
  qed
} note nonneg-all-proof = this

moreover
{ fix a b
  have        Rep-JavaInt
    (Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) * b +
    Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b)) =
    Rep-JavaInt a
    apply (cases 0 ≤ Rep-JavaInt a)
    apply (rule nonneg-all-proof)
    apply (simp add: zle-def)
    apply (rule nonpos-all-proof,auto)
    done
} note all-all-proof = this

moreover
{ fix a b
  have ⋀ z. ⟦ Rep-JavaInt a = z * Rep-JavaInt b;
    Rep-JavaInt b ≠ 0;
    Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1 ⟧
    ⟹ Rep-JavaInt (Int-to-JavaInt z * b) = z * Rep-JavaInt b
    apply (simp add: times-def)
    apply (subgoal-tac z ∈ JavaInt)
    apply (simp add: Int-to-JavaInt-ident)
    apply (drule HOL.eq-commute[THEN iffD1])
    apply (rotate-tac −1)
    apply (erule ssubst)
    apply (simp add: Int-to-JavaInt-ident Rep-JavaInt)
    apply (simp add: Factor-in-JavaInt)
    done
  } note LZ1 = this

moreover
{ fix a b
  assume 1: Rep-JavaInt a = z * Rep-JavaInt b
  assume 2: Rep-JavaInt b ≠ 0
  assume 3: Rep-JavaInt a ≠ MinInt-int | Rep-JavaInt b ≠ −1
  have Rep-JavaInt (Int-to-JavaInt z * b) = z * Rep-JavaInt b
    apply (simp add: times-def)
    apply (subgoal-tac z ∈ JavaInt)
    apply (simp add: Int-to-JavaInt-ident)
```

```
    apply (insert 1)
    apply (drule HOL.eq-commute[THEN iffD1])
    apply (rotate-tac −1)
    apply (erule ssubst)
    apply (simp add: Int-to-JavaInt-ident Rep-JavaInt)
    apply (insert 2 3)
    apply (simp add: Factor-in-JavaInt)
    done
  } note LZ1-noquant = this

moreover
have LZ2: ⋀z::int.
      ⟦ Rep-JavaInt a = z ∗ Rep-JavaInt b; Rep-JavaInt b ≠ (0::int) ⟧
      ⟹ Rep-JavaInt (Int-to-JavaInt z ∗ b) = z ∗ Rep-JavaInt b
  apply (case-tac Rep-JavaInt a = MinInt-int)
  apply (case-tac Rep-JavaInt b = −1)
  — solve problem for special case MinInt-int and −1::'a
  apply simp
  apply (rotate-tac 1)
  apply (simp add: HOL.eq-commute)
  apply (simp add: equation-zminus)

  apply (simp add: times-def)
  apply (simp add: Int-to-JavaInt-def MinInt-def Minustwo-Minus-two)
  — proved special case

  apply (rule LZ1, auto) +
  done

 moreover
  have SG1: 0 < a & b < 0
  ⟹ ((∀ z. Rep-JavaInt a ≠ z ∗ Rep-JavaInt b) ⟶
      Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
      b ∗ Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
      a) &
     ((EX z. Rep-JavaInt a = z ∗ Rep-JavaInt b) ⟶
      ((∀ z. Rep-JavaInt a ≠ z ∗ Rep-JavaInt b) ⟶
       Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
       b ∗ Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
       a) &
      Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
      b ∗ Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
      a)
   apply auto
   apply (subst Rep-JavaInt-inject [symmetric])
   apply (subst JavaInt-times-commute [of b -])
   apply (subst JavaInt-add-commute)
   apply (rule all-all-proof)
   apply (simp add: Rep-JavaInt-inject [symmetric]
```

```
                zero-def Abs-JavaInt-inverse)
    apply (simp add: zero-def[symmetric])
    apply (simp add: Rep-JavaInt-inject [symmetric])
    apply (simp add: LZ2)
    done

moreover
have SG2b: ⋀z::int.
      ⟦ Rep-JavaInt a = z * Rep-JavaInt b; Rep-JavaInt b ≠ (0::int) ⟧
      ⟹ Int-to-JavaInt z * b = a
  apply (simp add: Rep-JavaInt-inject [symmetric])
  apply (rule LZ2, auto)
  done

moreover
have SG2:  0 < a ⟶ ~ b < 0
   ⟹ ((0 < a & b < 0 | a < 0 & 0 < b) & (∀ z. Rep-JavaInt a ≠ z * Rep-JavaInt
b) ⟶
        Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
        b * Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
        a &
        ((a < 0 ⟶ ~ 0 < b) ⟶
         b + (Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
              b * Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b)) =
         a)) &
       ((a < 0 ⟶ 0 < b ⟶ (EX z. Rep-JavaInt a = z * Rep-JavaInt b)) ⟶
        ((0 < a & b < 0 | a < 0 & 0 < b) &
         (∀ z. Rep-JavaInt a ≠ z * Rep-JavaInt b) ⟶
         Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
         b * Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
         a) &
        Abs-JavaInt (Rep-JavaInt a mod Rep-JavaInt b) +
        b * Int-to-JavaInt (Rep-JavaInt a div Rep-JavaInt b) =
        a)
  apply (auto simp: eq-znull-eq-jnull)
  prefer 4
  apply (simp add: zero-def eq-znull-eq-jnull)
  prefer 8
  apply (simp add: zero-def eq-znull-eq-jnull)

  prefer 4
  apply (simp-all add: zero-def[symmetric])
  apply (rule SG2b,auto simp: eq-znull-eq-jnull[of b])
  prefer 7
  apply (rule SG2b, auto  simp: eq-znull-eq-jnull[of b])

  — now we have 6 equally shaped subgoals
  apply (simp-all only: JavaInt-times-commute [of b -])
   apply (subst JavaInt-add-commute,subst Rep-JavaInt-inject [symmetric], rule
```

```
all-all-proof)+
  done

  moreover — this is the main theorem – impossible to use ultimately ... directly

  have ((a::JavaInt) div b) * b + (a mod b) = a
    apply (simp only: div-def mod-def)
    apply (cases 0 < a & b < 0)
    apply (simp-all split: split-if)
    — 2 subgoals
    apply (rule SG1)
    prefer 2
    apply (rule SG2)
    apply auto
done

  ultimately show ?thesis by auto
qed
```

16.7 Calculating Example Values for div and mod

The following lemmas formalize the properties described in (6) for the modulo operator (JLS §15.17.3), i.e. they prove that the example values given in the JLS are indeed met by our formalization.

```
lemma div-mod-example1 :
  (5::JavaInt) mod 3 = 2
  apply (simp add: mod-def number-of-def)
  apply (simp add: JavaInt-less-def JavaInt-le-def zero-def diff-def)
  apply (subgoal-tac 5 ∈ JavaInt)
  apply (subgoal-tac 3 ∈ JavaInt)
  apply (subgoal-tac 2 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (thin-tac [1−2] ?x ∈ JavaInt)
  apply (thin-tac ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  done

lemma div-mod-example2 :
  (5::JavaInt) div 3 = 1
  apply (simp add: div-def number-of-def)
  apply (simp add: JavaInt-less-def JavaInt-le-def zero-def add-def one-def)
  apply (subgoal-tac 5 ∈ JavaInt)
  apply (subgoal-tac 3 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (thin-tac ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  done
```

```
lemma div-mod-example3 :
  (5::JavaInt) mod −3 = 2
  apply (simp add: mod-def number-of-def splits del: split-if)
  apply (simp add: JavaInt-less-def JavaInt-le-def add-def uminus-def
    zero-def splits del: split-if)
  apply (subgoal-tac 5 ∈ JavaInt)
  apply (subgoal-tac −3 ∈ JavaInt)
  apply (subgoal-tac 2 ∈ JavaInt)
  apply (subgoal-tac 3 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (simp add: Abs-JavaInt-inject)
  apply (simp add: number-of-def)
  apply (insert minusone-in-JavaInt)
  apply (simp only: Int-to-JavaInt-ident)
  apply (simp only: Abs-JavaInt-inject)
  apply simp

  apply (thin-tac [1−5] ?x ∈ JavaInt)
  apply (thin-tac [1−4] ?x ∈ JavaInt)
  apply (thin-tac [1−3] ?x ∈ JavaInt)
  apply (thin-tac [1−2] ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)

  apply (rule HOL.ccontr)
  apply simp
  apply (simp only: HOL.eq-commute [of - 5] zmult-commute [of - −3])
  apply (simp only: zmod-eq-0-iff[symmetric])
  apply auto
done

lemma div-mod-example4 :
  (5::JavaInt) div −3 = −1
  apply (simp add: div-def number-of-def)
  apply (simp add: JavaInt-less-def JavaInt-le-def zero-def one-def add-def)
  apply (subgoal-tac 5 ∈ JavaInt)
  apply (subgoal-tac −3 ∈ JavaInt)
  apply (subgoal-tac −2 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (simp add: Abs-JavaInt-inject)
  apply (simp add: number-of-def)
  apply (insert minusone-in-JavaInt)
  apply (simp only: Int-to-JavaInt-ident)
  apply (simp only: Abs-JavaInt-inject)
  apply (thin-tac [1−4] ?x ∈ JavaInt)
  apply (thin-tac [1−3] ?x ∈ JavaInt)
  apply (thin-tac [1−2] ?x ∈ JavaInt)
  apply (thin-tac ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  apply (rule HOL.ccontr)
```

```
  apply simp
  apply (simp only: HOL.eq-commute [of - 5] zmult-commute [of - −3])
  apply (simp only: zmod-eq-0-iff [symmetric])
  apply auto
done


lemma div-mod-example5 :
  (−5::JavaInt) mod 3 = −2
  apply (simp add: mod-def number-of-def)
  apply (simp add: JavaInt-less-def JavaInt-le-def zero-def add-def uminus-def)
  apply (subgoal-tac −5 ∈ JavaInt)
  apply (subgoal-tac 3 ∈ JavaInt)
  apply (subgoal-tac −3 ∈ JavaInt)
  apply (subgoal-tac −2 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (simp add: Abs-JavaInt-inject)
  apply (thin-tac [1−4] ?x ∈ JavaInt)
  apply (thin-tac [1−3] ?x ∈ JavaInt)
  apply (thin-tac [1−2] ?x ∈ JavaInt)
  apply (thin-tac ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  apply (rule HOL.ccontr)
  apply simp
  apply (simp only: HOL.eq-commute [of - −5] zmult-commute [of - 3])
  apply (simp only: zmod-eq-0-iff [symmetric])
  apply auto
done

lemma div-mod-example6 :
  (−5::JavaInt) div 3 = −1
  apply (simp add: div-def number-of-def)
  apply (simp add: JavaInt-less-def JavaInt-le-def zero-def add-def one-def)
  apply (subgoal-tac −5 ∈ JavaInt)
  apply (subgoal-tac 3 ∈ JavaInt)
  apply (subgoal-tac −2 ∈ JavaInt)
  apply (simp add:  Int-to-JavaInt-ident)
  apply (simp add: Abs-JavaInt-inject)
  apply (simp add: number-of-def)
  apply (insert minusone-in-JavaInt)
  apply (simp only: Int-to-JavaInt-ident)
  apply (simp only: Abs-JavaInt-inject)
  apply (thin-tac [!] ?x ∈ JavaInt)
  apply (thin-tac [1−3] ?x ∈ JavaInt)
  apply (thin-tac [1−2] ?x ∈ JavaInt)
  apply (thin-tac ?x ∈ JavaInt)
  apply (simp-all add: JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def)
  apply (rule HOL.ccontr)
  apply simp
```

apply (*simp only*: *HOL.eq-commute* [*of* - −*5*] *zmult-commute* [*of* - *3*])
apply (*simp only*: *zmod-eq-0-iff*[*symmetric*])
apply *auto*
done

lemma *div-mod-example7* :
(−*5*::*JavaInt*) *mod* −*3* = −*2*
apply (*simp add*: *mod-def number-of-def*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def zero-def add-def uminus-def*)
apply (*subgoal-tac* −*5* ∈ *JavaInt*)
apply (*subgoal-tac* −*3* ∈ *JavaInt*)
apply (*subgoal-tac* −*2* ∈ *JavaInt*)
apply (*simp add*: *Int-to-JavaInt-ident*)
apply (*thin-tac* [*1*−*2*] *?x* ∈ *JavaInt*)
apply (*thin-tac ?x* ∈ *JavaInt*)
apply (*simp-all add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)
done

lemma *div-mod-example8* :
(−*5*::*JavaInt*) *div* −*3* = *1*
apply (*simp add*: *div-def number-of-def*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def zero-def add-def one-def*)
apply (*subgoal-tac* −*5* ∈ *JavaInt*)
apply (*subgoal-tac* −*3* ∈ *JavaInt*)
apply (*simp add*: *Int-to-JavaInt-ident*)
apply (*thin-tac ?x* ∈ *JavaInt*)
apply (*simp-all add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)
done

The following test lemmas were added by us because we found it important to verify the behavior of div and mod in these cases.

lemma *div-mod-example9* :
(*8*::*JavaInt*) *div* −*2* = −*4*
apply (*simp add*: *div-def number-of-def*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def zero-def one-def add-def*)
apply (*subgoal-tac 8* ∈ *JavaInt*)
apply (*subgoal-tac* −*2* ∈ *JavaInt*)
apply (*subgoal-tac* −*4* ∈ *JavaInt*)
apply (*subgoal-tac* −*3* ∈ *JavaInt*)
apply (*simp add*: *Int-to-JavaInt-ident*)
apply (*simp add*: *Abs-JavaInt-inject*)
apply (*thin-tac* [*1*−*3*] *?x* ∈ *JavaInt*)
apply (*thin-tac* [*1*−*2*] *?x* ∈ *JavaInt*)
apply (*thin-tac ?x* ∈ *JavaInt*)
apply (*simp-all add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)
done

lemma *div-mod-example10* :
(*0*::*JavaInt*) *div* −*2* = *0*

apply (*simp add*: *div-def number-of-def*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def zero-def one-def add-def*)
done

lemma *div-mod-example11* :
(-8::*JavaInt*) *mod 12* = -8
apply (*simp add*: *mod-def number-of-def*)
apply (*simp add*: *JavaInt-less-def JavaInt-le-def zero-def add-def uminus-def*)
apply (*subgoal-tac* $-8 \in$ *JavaInt*)
apply (*subgoal-tac 12* $\in$ *JavaInt*)
apply (*subgoal-tac* $-12 \in$ *JavaInt*)
apply (*subgoal-tac 4* $\in$ *JavaInt*)
apply (*simp add*: *Int-to-JavaInt-ident*)
apply (*simp add*: *Abs-JavaInt-inject*)
apply (*thin-tac* [*1*$-$*4*] *?x* $\in$ *JavaInt*)
apply (*thin-tac* [*1*$-$*3*] *?x* $\in$ *JavaInt*)
apply (*thin-tac* [*1*$-$*2*] *?x* $\in$ *JavaInt*)
apply (*thin-tac ?x* $\in$ *JavaInt*)
apply (*simp-all add*: *JavaInt-def MinInt-int-def MaxInt-int-def BitLength-def*)
apply (*rule HOL.ccontr*)
apply *simp*
apply (*simp only*: *HOL.eq-commute* [*of* - $-$*2*] *zmult-commute* [*of* - *3*])
apply (*simp only*: *zmod-eq-0-iff*[*symmetric*])
apply *auto*
done

end

17 Formalization With Bitstring Representation

theory *JavaIntegersBit* = *JavaIntegersDef* :

lemma *numeral0-is-number-of-pls*:
(*number-of*::*bin*$\Rightarrow$*int*) *bin.Pls* = *Numeral0*
by (*auto*)

constdefs
bin-of :: *JavaInt* $\Rightarrow$ *bin*
bin-of x $\equiv$ *THE y*. *x* = *number-of y*

consts
pad :: [*nat*,*bin*] ⇒ *bin*

primrec
pad-0 : *pad 0 x* = *x*
pad-Suc : *pad* (*Suc k*) *x* =
(*case x of*
Pls ⇒ *pad k* (*bin.Pls BIT True*)
| *Min* ⇒ *pad k* (*bin.Min BIT False*)
| (*w BIT b*) ⇒ (*pad k w*) *BIT b*)

The function get_sign yields the sign (Pls or bin.Min) of the binary number passed as argument.

consts
get-sign :: *bin* ⇒ *bin*
primrec
get-sign-Pls : *get-sign bin.Pls* = *bin.Pls*
get-sign-Min : *get-sign bin.Min* = *bin.Min*
get-sign-BIT : *get-sign* (*w BIT b*) = (*get-sign w*)

The function trunc cuts off n bits from the right of the bitstring. It is equivalent to right shift with sign extension.

consts
trunc :: [*bin*, *nat*] ⇒ *bin*
primrec
trunc-0 : *trunc b 0* = *b*
trunc-Suc : *trunc b* (*Suc n*) =
(*case b of*
Pls ⇒ *bin.Pls*
| *Min* ⇒ *bin.Min*
| (*w1 BIT b1*) ⇒ (*trunc w1 n*))

The function trim b n keeps the lowest n bits of b and also keeps the sign.

consts
trim :: [*bin*,*nat*] ⇒ *bin*
primrec
trim-0 : *trim b 0* =
(*case b of*
Pls ⇒ *bin.Pls*
| *Min* ⇒ *bin.Min*
| (*w1 BIT b1*) ⇒ (*get-sign w1*)) — throw away the rest, keep only the sign
trim-Suc : *trim b* (*Suc n*) =
(*case b of*
Pls ⇒ *bin.Pls*
| *Min* ⇒ *bin.Min*
| (*b1 BIT w*) ⇒ (*trim b1 n*) *BIT w*)

False is equivalent to 0 in the Java bit representation.

The shift operators are not properly described in the JLS (§15.19) either. It is especially unclear what happens if the right-hand-side operand of the shift operators is negative.

The function shleft shifts the binary number left n bits, by inserting zeroes to the right of the number.

consts
  shleft :: [*bin*,*nat*] ⇒ *bin*
primrec
  shleft-0 : *shleft b 0* = *b*
  shleft-Suc : *shleft b* (*Suc n*) = (*shleft b n*) *BIT False*

The function insNull b n inserts an infinite number of False's at position n of the bitstring, counted from the right.

consts
  insNull :: [*bin*,*nat*] ⇒ *bin*
primrec
  insNull-0 : *insNull b 0* = *bin.Pls*
  insNull-Suc : *insNull b* (*Suc n*) =
        (*case b of*
         Pls ⇒ *bin.Pls*
        | *Min* ⇒ *bin.Pls*
        | (*b1 BIT w*) ⇒ (*insNull b n*) *BIT w*)

consts
  map-bin :: [*bool*⇒*bool*, *bin*] ⇒ *bin*

primrec
  map-bin-Pls : *map-bin f bin.Pls* = (*if f False then bin.Min else bin.Pls*)
  map-bin-Min : *map-bin f bin.Min* = (*if f True then bin.Min else bin.Pls*)
  map-bin-BIT : *map-bin f* (*w BIT x*) =
        (*map-bin f w*) *BIT* (*f x*)

The function zip_bin joins two bitstrings into one. The function passed as argument calculates the bits and thus determines which bitwise operation is performed on the bitstrings. We **do not** assume that both arguments are padded.

consts
  zip-bin :: [[*bool*,*bool*]⇒*bool*,*bin*,*bin*] ⇒ *bin*

primrec
zip-bin-Pls : *zip-bin f-bool x bin.Pls* =
    (*case x of*
      Pls ⇒ *if* (*f-bool False False*) *then bin.Min else bin.Pls*

$| \ Min \Rightarrow if \ (f\text{-}bool \ True \ False) \ then \ bin.Min \ else \ bin.Pls$
$| \ (w \ BIT \ b) \Rightarrow (map\text{-}bin \ (\lambda x. \ f\text{-}bool \ x \ False) \ w)$
$BIT \ (f\text{-}bool \ b \ False))$

$zip\text{-}bin\text{-}Min : zip\text{-}bin \ f\text{-}bool \ x \ bin.Min =$
$(case \ x \ of$
$Pls \Rightarrow if \ (f\text{-}bool \ False \ True) \ then \ bin.Min \ else \ bin.Pls$
$| \ Min \Rightarrow if \ (f\text{-}bool \ True \ True) \ then \ bin.Min \ else \ bin.Pls$
$| \ (w \ BIT \ b) \Rightarrow (map\text{-}bin \ (\lambda x. \ f\text{-}bool \ x \ True) \ w)$
$BIT \ (f\text{-}bool \ b \ True))$

$zip\text{-}bin\text{-}BIT : zip\text{-}bin \ f\text{-}bool \ x \ (w2 \ BIT \ b2) =$
$(case \ x \ of$
$Pls \Rightarrow (map\text{-}bin \ (f\text{-}bool \ False) \ w2) \ BIT \ (f\text{-}bool \ False \ b2)$
$| \ Min \Rightarrow (map\text{-}bin \ (f\text{-}bool \ True) \ w2) \ BIT \ (f\text{-}bool \ True \ b2)$
$| \ (w1 \ BIT \ b1) \Rightarrow$
$(zip\text{-}bin \ f\text{-}bool \ w1 \ w2) \ BIT \ (f\text{-}bool \ b1 \ b2) \ )$

17.1 Bitwise Operators - AND, OR, XOR

This section formalizes the bitwise AND, OR, and exclusive OR operators.

Java Language Specification [GJSB00], §15.22, 15.22.1

"The bitwise operators [...] include the AND operator &, exclusive OR operator ˆ, and inclusive OR operator |.(1) [...]
Each operator is commutative if the operand expressions have no side effects. Each operator is associative.(2) [...]
For &, the result value is the bitwise AND of the operand values. For ˆ, the result value is the bitwise exclusive OR of the operand values. For |, the result value is the bitwise inclusive OR of the operand values. For example, the result of the expression 0xff00 & 0xf0f0 is 0xf000. The result of 0xff00 ˆ 0xf0f0 is 0x0ff0. The result of 0xff00 | 0xf0f0 is 0xfff0.(3)"

These bitwise operators are formalized as follows:

constdefs
$JavaInt\text{-}bitand :: [JavaInt, JavaInt] \Rightarrow JavaInt \quad (\textbf{infixl} \ \&\# \ 60)$
$x \ \&\# \ y \equiv$
$number\text{-}of \ (zip\text{-}bin \ (op \ \&::[bool,bool]\Rightarrow bool) \ (bin\text{-}of \ x) \ (bin\text{-}of \ y))$

$JavaInt\text{-}bitxor :: [JavaInt, JavaInt] \Rightarrow JavaInt \quad (\textbf{infixl} \ \hat{}\# \ 59)$
$x \ \hat{}\# \ y \equiv$
$number\text{-}of \ (zip\text{-}bin \ (\lambda \ (x::bool) \ y. \ x \neq y) \ (bin\text{-}of \ x) \ (bin\text{-}of \ y))$

$JavaInt\text{-}bitor :: [JavaInt, JavaInt] \Rightarrow JavaInt \quad (\textbf{infixl} \ |\# \ 58)$
$x \ |\# \ y \equiv$
$number\text{-}of \ (zip\text{-}bin \ (op \ |::[bool,bool]\Rightarrow bool) \ (bin\text{-}of \ x) \ (bin\text{-}of \ y))$

where bin-of transforms a JavaInt into its bitstring representation, zip-bin merges two bitstrings into one by applying a function (which is passed as the first argument) to each bit pair in turn, and number-of turns the resulting bitstring back into a JavaInt.

lemma *trim-True* [*simp*] :
  ((*number-of*(*bin.Pls BIT False BIT x*))::*int*) = *number-of*(*bin.Pls BIT x*)
  apply (*simp add*: *number-of-def*)
  apply (*cases x*)
  apply (*auto*)
  apply (*simp-all only*: *iszero-def*)
  done

lemma *trim-False* [*simp*] :
  ((*number-of*(*bin.Min BIT True BIT x*))::*int*) = *number-of*(*bin.Min BIT x*)
  apply (*simp add*: *number-of-def*)
  apply (*cases x*)
  apply (*auto*)
  apply (*simp-all only*: *iszero-def*)
  done

The commutativity of the three operators is first proven generically, which makes the concrete proofs for the three operators trivial:

lemma *zip-bin-commute*:
($\bigwedge$ *x y*. ((*f x y*) = (*f y x*))) $\Longrightarrow$ ((*zip-bin f a b*) = (*zip-bin f b a*))
proof −
  have $*$: ($\bigwedge$ *x y*. (*f x y*) = (*f y x*)) $\Longrightarrow$ (λ *x*. *f x False*) = *f False*)
  by (*rule ext*, *auto*)
  moreover
  have $**$: ($\bigwedge$ *x y*. (*f x y*) = (*f y x*)) $\Longrightarrow$ (λ *x*. *f x True*) = *f True*)
  by (*rule ext*, *auto*)

assume *1*: ($\bigwedge$ *x y*. ((*f x y*) = (*f y x*)))
with *1* **show** ((*zip-bin f a b*) = (*zip-bin f b a*))
  apply (*rule-tac x* = *b* **in** *spec*)
  apply (*induct rule*: *bin.induct*) — induction over a

  apply (*simp-all add*: $*$ $**$ *split add*: *bin.split*)
  done
qed

lemma *bitand-commute*:
a &# *b* = *b* &# *a*
by (*simp add*: *JavaInt-bitand-def*, *subst zip-bin-commute*, *auto*)

lemma *bitxor-commute*:
a ^# *b* = *b* ^# *a*
by (*simp add*: *JavaInt-bitxor-def*, *subst zip-bin-commute*, *auto*)

lemma *bitor-commute*:
a |# *b* = *b* |# *a*
by (*simp add*: *JavaInt-bitor-def*, *subst zip-bin-commute*, *auto*)

end

18 Formalizing the Exceptional Behavior Java Integers

theory *JavaIntegers* = *JavaIntegersDiv* + *Lifting*:

The Java Language Specification introduces the concept of *exception* in expressions and statements of the language:

> Java Language Specification [GJSB00], §11.3, §11.3.1
>
> "The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception [...]
> when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated."

Thus, exceptions have two aspects in Java:

- they change the control flow of a program,
- they are a particular kind of side-effect (i.e. an exception object is created), and they prevent program parts from having side-effects.

While we deliberately neglect the latter aspect in our model (which can be handled in a Hoare Calculus on full Java, for example, when integrating our expression language into the statement language), we have to cope with the former aspect since it turns out to have dramatic consequences for the rules over Java expressions (these effects have not been made precise in the JLS).

So far, our normal behavior model is a completely denotational model; each expression is assigned a value by our semantic definitions. We maintain this denotational view, with the consequence that we have to introduce *exceptional values* that are assigned to expressions that "may [not] appear to have been evaluated". In the language fragment we are considering, only one kind of exception may occur:

Java Language Specification [GJSB00], §4.2.2

"The only numeric operators that can throw an exception (§11) are the integer divide operator / (§15.17.2) and the integer remainder operator % (§15.17.3), which throw an ArithmeticException if the right-hand operand is zero."

In order to achieve a clean separation of concerns, we apply the technique developed in [BW03]. Conceptually, a theory morphism is used to convert a normal behavior model into a model enriched by exceptional behavior. Technically, the effect is achived by redefining all operators such as $+,-,*$ etc. using "semantical wrapper functions" and the normal behavior definitions given in the previous chapters. Two types of theory morphisms can be distinguished: One for a *one-exception world*, the other for a *multiple-exception world*. While the former is fully adequate for the arithmetic language fragment we are discussing throughout this report, the latter is the basis for future extensions by e.g. array access constructs which may raise *out-of-bounds exceptions*. In the following, we therefore present the former in more detail and only outline the latter.

18.1 The One-Exception Theory Morphism

We begin with the introduction of a type constructor that disjointly adds to a type α a failure element such as $\perp$ (see e.g. [Win93], where the following construction is also called "lifting"). We declare a type class *bot* for all types containing a failure element $\perp$ and define as *semantical combinator*, i.e. as "wrapper function" of this theory morphism, the combinator *strictify* that turns a function into its *strict extension* wrt. the failure elements:

strictify $::['a \Rightarrow 'b,\ 'a] \Rightarrow 'b$

$\mathit{strictify}\ ?f\ ?x \equiv \mathit{if}\ ?x = \mathit{UU}\ \mathit{then}\ \mathit{UU}\ \mathit{else}\ ?f\ ?x$

Moreover, we introduce the definedness predicate DEF $::'a \Rightarrow \mathit{bool}$ by $\mathit{DEF}\ ?x \equiv ?x \neq \mathit{UU}$.

Now we introduce a concrete type constructor that lifts any type α into the type class bot:

$$\textbf{datatype}\quad \mathrm{up}(\alpha) = \lfloor\_\rfloor\ \alpha\ \mid\ \perp$$

In the sequel, we write $t_\perp$ instead of up(t). We define the inverse to the constructor $\lfloor\_\rfloor$ as $\lceil\_\rceil$. Based on this infrastructure, we can now define the type $java_int$ that includes a failure element:

types *java-int* = *JavaInt up*

arities *up* :: (*zero*) *zero*
arities *up* :: (*one*) *one*

defs (**overloaded**)
zero-def : $0 \equiv \lfloor\ 0\ \rfloor$
one-def : $1 \equiv \lfloor\ 1\ \rfloor$

constdefs
minint :: *java-int*
minint $\equiv \lfloor$ *MinInt* $\rfloor$
maxint :: *java-int*
maxint $\equiv \lfloor$ *MaxInt* $\rfloor$

arities *up* :: (*plus*) *plus*
arities *up* :: (*minus*) *minus*
arities *up* :: (*times*) *times*
arities *up* :: (*Divides.div*) *Divides.div*

Furthermore, we can now define the operations on this enriched type; e.g. we convert the *JavaInt* unary minus operator into the related *java-int* operatory (note that Isabelle supports overloading):

defs (**overloaded**)
plus-def : $op + \equiv$ *strictify* (λ *X*.
strictify (λ *Y*.
$\lfloor$ (*Lifting.drop X*) + (*Lifting.drop Y*) $\rfloor$))

defs (**overloaded**)
uminus-def : *uminus* $\equiv$ *strictify* (λy. $\lfloor$ ($-$ (*Lifting.drop y*)) $\rfloor$)

defs (**overloaded**)
minus-def : $op - \equiv$ *strictify* (λ *X*.
strictify (λ *Y*.
$\lfloor$ (*Lifting.drop X*) $-$ (*Lifting.drop Y*) $\rfloor$))

defs (**overloaded**)
abs-def : *abs* $\equiv$ *strictify*(λX. $\lfloor$ *abs* (*Lifting.drop X*) $\rfloor$)

defs (**overloaded**)
times-def : $op * \equiv$ *strictify* (λ *X*.
strictify (λ *Y*.
$\lfloor$ (*Lifting.drop X*) $*$ (*Lifting.drop Y*) $\rfloor$))

defs (**overloaded**)
div-def : *op div* ≡ *strictify* (λ *X*.
strictify (λ *Y*.
if (*Lifting.drop Y*) = *0 then UU*
else ⌊(*Lifting.drop X*) *div* (*Lifting.drop Y*)⌋))

defs (**overloaded**)
mod-def : *op mod* ≡ *strictify* (λ *X*.
strictify (λ *Y*.
if (*Lifting.drop Y*) = *0 then UU*
else ⌊ (*Lifting.drop X*) *mod* (*Lifting.drop Y*) ⌋))

All binary arithmetic operators that are strict extensions like − or ∗ are constructed analogously; the equality and the logical operators like the strict logical AND & follow this scheme as well. For the division and modulo operators / and %, we add case distinctions whether the divisor is zero (yielding ⊥). Java's non-strict logical AND && is defined in our framework by explicit case distinctions for ⊥.

This adds new rules like X + ⊥= ⊥ and ⊥+ X = ⊥.

18.2 Proof of Basic Definedness

lemma *zero-DEF*[*simp*] : *DEF* (*0*::*java-int*)
by (*simp add*: *zero-def*)

lemma *one-DEF*[*simp*] : *DEF* (*1*::*java-int*)
by (*simp add*: *one-def*)

lemma *maxint-DEF*[*simp*] : *DEF* (*maxint*)
by (*simp add*: *maxint-def*)

lemma *minint-DEF*[*simp*] : *DEF* (*minint*)
by (*simp add*: *minint-def*)

18.3 Proof of exception propagation rules

lemma *uminus-UU*[*simp*] : −(*UU*::(′*a*::*minus*)*up*) = *UU*
by (*simp add*: *uminus-def*)

lemma *minus-UU1*[*simp*] : (*UU* − (*X*::(′*a*::*minus*)*up*)) = *UU*
by (*simp add*: *minus-def*)

lemma *minus-UU2*[*simp*] : ((*X*::(′*a*::*minus*)*up*) − *UU*) = *UU*
by (*simp add*: *minus-def*)

lemma *add-UU1*[*simp*] : (*UU* + (*X*::(′*a*::*plus*)*up*)) = *UU*
by (*simp add*: *plus-def*)

lemma *add-UU2*[*simp*] : $((X::('a::plus)up) + UU) = UU$
by (*simp add*: *plus-def*)

lemma *times-UU1*[*simp*] : $(UU * (X::('a::times)up)) = UU$
by (*simp add*: *times-def*)

lemma *times-UU2*[*simp*] : $((X::('a::times)up) * UU) = UU$
by (*simp add*: *times-def*)

lemma *div-UU1*[*simp*] : $(UU\ div\ (X::('a::\{Divides.div,zero\})up)) = UU$
by (*simp add*: *div-def*)

lemma *div-UU2*[*simp*] : $((X::('a::\{Divides.div,zero\})up)\ div\ UU) = UU$
by (*simp add*: *div-def*)

lemma *mod-UU1*[*simp*] : $(UU\ mod\ (X::('a::\{Divides.div,zero\})up)) = UU$
by (*simp add*: *mod-def*)

lemma *mod-UU2*[*simp*] : $((X::('a::\{Divides.div,zero\})up)\ mod\ UU) = UU$
by (*simp add*: *mod-def*)

lemma *div-0*[*simp*] : $((X::('a::\{Divides.div,zero\})up)\ div\ 0) = UU$
by (*case-tac DEF X*,*auto*,*simp add*: *div-def zero-def*)

lemma *mod-0*[*simp*] : $((X::('a::\{Divides.div,zero\})up)\ mod\ 0) = UU$
by (*case-tac DEF X*,*auto*,*simp add*: *mod-def zero-def*)

18.4 Lifting Axiomatic Classes to Versions with Exceptional Behaviour

But what happens with the properties established for the normal behavior semantics? They can also be lifted, and this process can even be automated (see [BW03] for details). Thus, the commutativity and associativity laws for normal behavior, e.g. $(X::JavaInt) + (Y::JavaInt) = Y + X$, can be lifted to $(X::JavaInt\ up) + (Y::JavaInt\ up) = Y + X$ by generic proof procedure establishing the case distinctions for failures. However, this works smoothly only if all variables occur on both sides of the equation; variables only occurring on one side have to be restricted to be defined. Consequently, the lifted version of the division theorem requires definedness of the variable that only occurs on one side of the equation; see below.

lemma *JavaInt-add-commute2*:
$(X::('a::plus\text{-}ac0)up) + Y = Y + X$
apply (*case-tac DEF X*, *case-tac DEF Y*,*auto*)
apply (*simp add*: *DEF-X-up plus-def plus-ac0.commute*)
done

lemma *JavaInt-add-assoc2*:
$(X::('a::plus\text{-}ac0)up) + Y + Z = X + (Y + Z)$
apply (*case-tac DEF X*, *case-tac DEF Y*,*case-tac DEF Z*, *auto*)
apply (*simp add*: *DEF-X-up plus-def plus-ac0.assoc*)
done

lemma *JavaInt-add-zero2*:
$0 + (X::('a::plus\text{-}ac0)up) = X$
apply (*case-tac DEF X*, *auto*)
apply (*simp add*: *plus-def zero-def plus-ac0.zero DEF-X-up*)
apply (*auto*)
done

lemma *JavaInt-minus-def2*:
$(X::('a::ring)up) - Y = X + (- Y)$
apply (*case-tac DEF X*, *case-tac DEF Y*,*auto*)
apply (*simp add*: *DEF-X-up plus-def minus-def uminus-def*)
done

lemma *JavaInt-l-neg2*:
$DEF(X::('a::ring)up) \Longrightarrow (-X) + X = 0$
apply (*simp add*: *DEF-X-up plus-def uminus-def zero-def ring.l-neg*)
done

lemma *JavaInt-times-unit2*:
$1 * (X::('a::ring)up) = X$
apply (*case-tac DEF X*, *auto*)
apply (*simp add*: *times-def one-def DEF-X-up*)
apply (*auto*)
done

lemma *JavaInt-times-commute2*:
$(X::('a::ring)up) * Y = Y * X$
apply (*case-tac DEF X*, *case-tac DEF Y*,*auto*)
apply (*simp add*: *DEF-X-up times-def ring.m-comm*)
done

lemma *JavaInt-times-assoc2*:
$(X::('a::ring)up) * Y * Z = X * (Y * Z)$
apply (*case-tac DEF X*, *case-tac DEF Y*,*case-tac DEF Z*, *auto*)
apply (*simp add*: *DEF-X-up times-def ring.m-assoc*)
done

lemma *JavaInt-l-distr2*:
$((X::('a::ring)up) + Y) * Z = X * Z + Y * Z$
apply (*case-tac DEF X*, *case-tac DEF Y*,*case-tac DEF Z*, *auto*)
apply (*simp add*: *times-def plus-def ring.l-distr DEF-X-up*)

done

instance *up* :: (*plus-ac0*)*plus-ac0*
apply *intro-classes*
apply (*auto intro*: *JavaInt-add-commute2 JavaInt-add-assoc2 JavaInt-add-zero2*)
done

instance *up* :: (*power*)*power* **..**

defs (**overloaded**)
power-def : (*a*::(′*a*::*power*) *up*) ^ *n* ≡
(*strictify* (λ *X*. ⌊ (*Lifting.drop X*) ^ *n* ⌋)*a*)

instance *up* :: (*inverse*)*inverse* **..**

When trying to add the lifted type to the axiomatic class *ring*, we discover a problem: technically, ′*a up* is not a ring. The main problem is l_neg: − (*a*::′*a up*) + *a* = (*0*::′*a up*) which simply does not hold in an exception behaviour class, only a weaker form: *DEF* (*a*::′*a up*) ⟹ − *a* + *a* = (*0*::′*a up*) (JavaInt_l_neg2). Similar problems arise for (*a* + *b* = *a* + *c*) = (*b* = *c*) and (*b* + *a* = *c* + *a*) = (*b* = *c*) which hold in Isabelle 2004 for the axiomatic class semiring, and

(*a*::′*a up*) *dvd* (*1*::′*a up*) ⟹ *inverse a* = (*0*::′*a up*) which did not make it into Isabelle04.

There is a big advantage for Isabelle2004 here: the organization into classes is much more fine-grained and therefore the damage is smaller. In Isabelle 2004, this structure still fits in "almost_semiring".

lemma *DEF-times* [*simp*] : *DEF*((*X*::*java-int*) ∗ *Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto simp*: *zero-def*)
apply (*simp-all add*: *times-def*)
done

lemma *DEF-plus* [*simp*] : *DEF*((*X*::*java-int*) + *Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto simp*: *zero-def*)
apply (*simp-all add*: *plus-def*)
done

lemma *DEF-minus* [*simp*] : *DEF*((*X*::*java-int*) − *Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto simp*: *zero-def*)
apply (*simp-all add*: *minus-def*)
done

lemma *DEF-uminus* [*simp*] : *DEF*(−(*X*::*java-int*)) = (*DEF X*)
apply (*cases DEF X*, *auto simp*: *zero-def*)

apply (*simp-all add*: *uminus-def*)
done

lemma *DEF-div-EQ* [*simp*] : *DEF*((*X*::*java-int*) *div Y*) = (*DEF X* & *DEF Y* & *Y* ≠ *0*)
apply (*cases DEF X*, *cases DEF Y*, *auto simp*: *zero-def*)
apply (*simp-all add*: *div-def*)
apply (*auto simp*: *DEF-X-up*)
done

lemma *DEF-mod-EQ* [*simp*] : *DEF*((*X*::*java-int*) *mod Y*) = (*DEF X* & *DEF Y* & *Y* ≠ *0*)
apply (*cases DEF X*, *cases DEF Y*, *auto simp*: *zero-def*)
apply (*simp-all add*: *mod-def*)
apply (*auto simp*: *DEF-X-up*)
done

18.5 Lifting Number Representations and Bridging to Computations

instance *up* :: (*number*)*number* **..**

defs (**overloaded**)
number-of-def : (*number-of*:: *bin* ⇒ ((′*a*::*number*)*up*)) *b* ≡ ⌊ (*number-of b*) ⌋

Computation rules:

lemma *uminus-compute*[*simp*] : −(*number-of X*) = (*?X*::*java-int*)
by (*simp add*: *uminus-def number-of-def*)

lemma *add-compute*[*simp*] : (*number-of X*) + (*number-of Y*) = (*?X*::*java-int*)
by (*simp add*: *plus-def number-of-def*)

lemma *times-compute*[*simp*] : (*number-of X*) ∗ (*number-of Y*) = (*?X*::*java-int*)
by (*simp add*: *times-def number-of-def*)

lemma *minus-compute*[*simp*] : (*number-of X*) − (*number-of Y*) = (*?X*::*java-int*)
by (*simp add*: *minus-def number-of-def*)

lemma *div-compute*[*simp*] : (*number-of X*) *div* (*number-of Y*) =
(*if* ((*number-of Y* − *MinInt-int*) *mod* −
(*2* ∗ *MinInt-int*) + *MinInt-int*) = *0*
then UU::*java-int*
else ⌊ (*number-of X*) *div* (*number-of Y*) ⌋)
apply(*simp only*: *div-def number-of-def Int-to-JavaInt-def*
JavaIntegersDef.zero-def strict22-DEF DEF-lift drop-lift)

apply(*subgoal-tac* (*number-of Y* = *Abs-JavaInt* (*0*::*int*)) =
((*number-of Y* − *MinInt-int*) *mod* − (*2*∗*MinInt-int*)+*MinInt-int*
=*0*))
apply *simp*
apply (*simp add*: *JavaIntegersDef*.*number-of-def Int-to-JavaInt-def*)
apply (*subst Abs-JavaInt-inject*)
apply (*simp-all*)
apply (*insert Int-to-JavaInt-charn*)
apply (*simp add*: *Int-to-JavaInt-def Rep-Abs-remove-general*)
done

lemma *mod-compute*[*simp*] : (*number-of X*) *mod* (*number-of Y*) =
(*if* ((*number-of Y* − *MinInt-int*) *mod* −
(*2* ∗ *MinInt-int*) + *MinInt-int*) = *0*
then UU::*java-int*
else ⌊ (*number-of X*) *mod* (*number-of Y*) ⌋)
apply(*simp only*: *mod-def number-of-def Int-to-JavaInt-def*
JavaIntegersDef.*zero-def strict22-DEF DEF-lift drop-lift*)
apply(*subgoal-tac* (*number-of Y* = *Abs-JavaInt* (*0*::*int*)) =
((*number-of Y* − *MinInt-int*) *mod* − (*2*∗*MinInt-int*)+*MinInt-int*
=*0*))
apply *simp*
apply (*simp add*: *JavaIntegersDef*.*number-of-def Int-to-JavaInt-def*)
apply (*subst Abs-JavaInt-inject*)
apply (*simp-all*)
apply (*insert Int-to-JavaInt-charn*)
apply (*simp add*: *Int-to-JavaInt-def Rep-Abs-remove-general*)
done

lemma *ex1*: *5 div 3* = (*1*::*java-int*)
apply (*simp add*: *div-mod-example2 one-def MinInt-int-def BitLength-def*)
done

18.6 Lifting JavaInteger Theorems

lemma *JavaInt-div-mod-result* :
⟦ *DEF Y*; *Y* ≠ *0* ⟧ ⟹ ((*X*::*java-int*) *div Y*) ∗ *Y* + (*X mod Y*) = *X*
apply (*case-tac DEF X*,*auto*)
apply (*simp add*: *DEF-X-up div-def mod-def times-def plus-def zero-def*, *auto*)
apply (*subst JavaInt-times-commute*)
apply (*rule JavaInt-div-mod-result*)
done

lemma *JavaInt-0-div*[*simp*]: ⟦ *DEF X*; *X* ≠ *0*⟧ ⟹ *0 div X* = (*0*::*java-int*)
by (*simp add*: *DEF-X-up div-def mod-def times-def plus-def zero-def*, *auto*)

lemma *minint-div-minusone*[*simp*]:*minint div* -1 = *minint*
apply (*simp add*: *DEF-X-up div-def number-of-def*
minint-def MinInt-div-minusone)
apply (*simp add*:*JavaIntegersDef.number-of-def JavaIntegersDef.zero-def*)
apply (*subst Int-to-JavaInt-ident*)
defer *1*
apply (*subst Abs-JavaInt-inject*)
apply (*simp-all*)
done

lemma *diff-self*[*simp*]:⟦ *DEF a*; *a* ≠ *0* ⟧ ⟹ *a div a* = (*1*::*java-int*)
by (*simp add*: *DEF-X-up div-def zero-def one-def*, *auto*)

lemma *mod-self*[*simp*]: ⟦ *DEF a*; *a* ≠ *0* ⟧ ⟹ *a mod a* = (*0*::*java-int*)
by (*simp add*: *DEF-X-up mod-def zero-def one-def*, *auto*)

lemma *mod-divides*[*simp*]:
assumes *1*: *DEF a*
assumes *2*: *DEF b*
assumes *3*: (*Rep-JavaInt*(*Lifting.drop a*)) *mod* (*Rep-JavaInt* (*Lifting.drop b*))
= *0*
— i.e. representation a divides representation b
assumes *4*: *b* ≠ *0*
shows *a mod b* = (*0*::*java-int*)
apply (*insert 1 2 3 4*)
apply (*simp add*: *DEF-X-up mod-def zero-def one-def*, *auto*)
apply (*rule JavaIntegersDiv.JavaInt-mod-0*, *auto*)
done

lemma *minint-mod-minusone*[*simp*]: *minint mod* -1 = *0*
apply (*simp add*: *DEF-X-up mod-def minint-def zero-def number-of-def one-def*,*auto*)
apply (*simp add*: *JavaIntegersDef.number-of-def JavaIntegersDef.zero-def*)

apply (*rule swap* [*of Int-to-JavaInt* (-1::*int*) = *Abs-JavaInt* (*0*::*int*)])
apply (*subst Int-to-JavaInt-ident*)
prefer *2*
apply (*subst Abs-JavaInt-inject*, *auto*)
done

lemma *div-neg-shift*:
assumes *1*:*a*≠*minint*
assumes *2*:*b*≠*minint*

shows $a\ div\ (-b) = (-a)\ div\ b$
apply (*case-tac DEF a*,*auto*)
apply (*case-tac DEF b*,*auto*)
apply (*insert 1 2*)
apply (*simp add*: *DEF-X-up div-def minint-def zero-def*
number-of-def one-def uminus-def JavaInt-div-neg-shift,*auto*)
apply (*rule JavaInt-div-neg-shift*, *auto*)
done

lemma *mod-neg1*:
assumes *1*:$a \neq minint$
assumes *2*:$b \neq minint$
shows $-(a\ mod\ b) = (-a)\ mod\ b$
apply (*case-tac DEF a*,*auto*)
apply (*case-tac DEF b*,*auto*)
apply (*insert 1 2*)
apply (*simp add*: *DEF-X-up mod-def minint-def zero-def*
number-of-def one-def uminus-def, *auto*)

apply(*rule JavaInt-mod-neg1*,*auto*)
done

lemma *mod-neg2*:
shows $a\ mod\ (-b) = (a{::}java\text{-}int)\ mod\ b$
apply (*case-tac DEF a*,*auto*)
apply (*case-tac DEF b*,*auto*)
apply (*simp add*: *DEF-X-up mod-def minint-def zero-def*
number-of-def one-def uminus-def, *auto*)

apply(*rule JavaInt-mod-neg2*)
done

18.7 The Multiple-Exception Theory Morphism

We briefly consider the case of a semantics with several different exceptions, e.g. array access which can possibly lead to out-of-bounds exceptions. Such a change of the model can be achieved by exchanging the theory morphism, leaving the normal behavior model unchanged.

It suffices to present the differences to the previous theory morphism here. Instead of the class bot we introduce the class exn requiring a family of undefined values $\bot_e$. The according type constructor is defined as:

$$\textbf{datatype}\quad \mathrm{up}(\alpha) = \lfloor\_\rfloor\ \alpha\ \mid\ \bot\ \text{exception}$$

and analogously to $\lceil\_\rceil$ we define exn-of($\bot_e$) $= e$ as the inverse of the constructor $\bot$; exn-of is defined by an arbitrary but fixed HOL-value *arbitrary* for exn-of($\lfloor\_\rfloor$) $=$ *arbitrary*. Definedness is DEF(x) $= (\forall e.x \neq \bot_e)$.

The definition of operators is analogous to the previous section for the canonical cases; and the resulting lifting as well. Note, however, that the lifting of the commutativity laws fails and has to be restricted to the following:

$$\llbracket\ \text{DEF X} = \text{DEF Y} \wedge \text{exn-of X} = \text{exn-of Y}\ \rrbracket \\ \Longrightarrow (\text{X}::\ \mathsf{JAVAINT}) + \text{Y} = \text{Y} + \text{X}$$

These restrictions caused by the lifting reflect the fact that commutativity does not hold in a multi-exception world; if the expression on the left-hand side does not raise the same exception as the one on the right-hand side, the expression order cannot be changed.

Of course, this problem could also be overcome by defining a congruence that identifies all exceptions; however, this complicates matters substantially.

We refrain from such a refined model considering different exceptions in this work. However, we emphasize that our proposed technique to use a theory morphism not only leads to a clear separation of concerns in the semantic description of Java, but also to the systematic introduction of the side-conditions of arithmetic laws in Java that are easily overlooked.

end

theory *JavaArith* = *JavaIntegers* + *Lifting* :

types *java-bool* = *bool up*

constdefs

TRUE :: *java-bool*
TRUE ≡ ⌊ *True* ⌋

FALSE :: *java-bool*
FALSE ≡ ⌊ *False* ⌋

lemma *DEF-TRUE* [*simp*]: *DEF TRUE*
by (*auto simp*:*TRUE-def*)

lemma *DEF-FALSE* [*simp*]: *DEF FALSE*

by (*auto simp*:*FALSE-def*)

constdefs

```
NOT      :: java-bool ⇒ java-bool
NOT      ≡ strictify(Lifting.lift o Not o Lifting.drop)
AND      :: [java-bool, java-bool] ⇒ java-bool
AND      ≡ strictify (λ X.
               strictify (λ Y.
                   ⌊ (Lifting.drop X) & (Lifting.drop Y) ⌋ ))

NEG      :: java-int ⇒ java-bool
NEG      ≡ strictify (λ X. ⌊ neg (Rep-JavaInt(Lifting.drop X)) ⌋ )

LE       :: [java-int, java-int] ⇒ java-bool
LE       ≡ strictify (λ X.
               strictify (λ Y.
                   ⌊ (Lifting.drop X) ≤ (Lifting.drop Y) ⌋ ))

LESS     :: [java-int, java-int] ⇒ java-bool
LESS     ≡ strictify (λ X.
               strictify (λ Y.
                   ⌊ (Lifting.drop X) < (Lifting.drop Y) ⌋ ))
```

lemma *NOT-UU*[*simp*] : (*NOT UU*) = *UU*
by (*simp add*: *NOT-def*)

lemma *NEG-UU*[*simp*] : (*NEG UU*) = *UU*
by (*simp add*: *NEG-def*)

lemma *abs-UU*[*simp*] : (*abs* (*UU*::*java-int*)) = *UU*
by (*simp add*: *abs-def*)

lemma *AND-UU1*[*simp*] : (*AND UU X*) = *UU*
by (*simp add*: *AND-def*)

lemma *AND-UU2*[*simp*] : (*AND X UU*) = *UU*
by (*simp add*: *AND-def*)

lemma *LE-UU1*[*simp*] : (*LE UU X*) = *UU*
by (*simp add*: *LE-def*)

lemma *LE-UU2*[*simp*] : (*LE X UU*) = *UU*
by (*simp add*: *LE-def*)

lemma *LESS-UU1*[*simp*] : (*LESS UU X*) = *UU*
by (*simp add*: *LESS-def*)

lemma *LESS-UU2*[*simp*] : (*LESS X UU*) = *UU*
by (*simp add*: *LESS-def*)

A bit of strict logic ...

lemma *DEF-isTRUE*[*simp*] : *X* = *TRUE* $\Longrightarrow$ *DEF X*
by *simp*

lemma *DEF-isFALSE*[*simp*] : *X* = *FALSE* $\Longrightarrow$ *DEF X*
by *simp*

lemma *DEF-isTRUE-I* [*intro*]: $[\![$ *DEF X*; *Lifting.drop X* $]\!] \Longrightarrow$ *X* = *TRUE*
by(*auto simp*: *DEF-X-up TRUE-def*)

lemma *DEF-isFALSE-I* : $[\![$ *DEF X*; ~ *Lifting.drop X* $]\!] \Longrightarrow$ *X* = *FALSE*
by(*auto simp*: *DEF-X-up FALSE-def*)

lemma *DEF-LE-EQ* [*simp*] : *DEF*(*LE X Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto*)
apply (*simp add*: *LE-def*)
done

lemma *DEF-LESS-EQ* [*simp*] : *DEF*(*LESS X Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto*)
apply (*simp add*: *LESS-def*)
done

lemma *DEF-AND-EQ* [*simp*] : *DEF*(*AND X Y*) = (*DEF X* & *DEF Y*)
apply (*cases DEF X*, *cases DEF Y*, *auto*)
apply (*simp add*: *AND-def*)
done

lemma *DEF-NEQ-EQ* [*simp*] : *DEF*(*NEG X*) = (*DEF X*)
by (*cases DEF X*, *auto*, *simp add*: *NEG-def*)

lemma *DEF-NOT-EQ* [*simp*] : *DEF*(*NOT X*) = (*DEF X*)
by (*cases DEF X*, *auto*, *simp add*: *NOT-def*)

lemma *DEF-abs-EQ* [*simp*] : *DEF*(*abs* (*X*::*java-int*)) = (*DEF X*)
by (*cases DEF X*, *auto*, *simp add*: *abs-def*)

18.8 Lifting orders and related theorems.

Here, there are many choices to lift the premises. They can be formalized in terms of the mathematical integers, normal behaviour or exceptional behaviour JavaIntegers. As a guideline, we chose the latter option since one may want to express these entities in terms of a programming logic in Java.

```
lemma quotient-sign-plus :
  assumes 0: DEF d & DEF n
  assumes 1: abs (Rep-JavaInt(Lifting.drop d)) <= abs(Rep-JavaInt(Lifting.drop
n))
  assumes 2: neg (Rep-JavaInt (Lifting.drop n)) = neg (Rep-JavaInt (Lifting.drop
d))
  assumes 3: n ≠ minint | d ≠ (− 1)
  assumes 4: d ≠ 0
  shows LESS 0 (n div d) = TRUE
  apply (insert 0 1 2 3 4)
  apply (simp add: DEF-X-up div-def zero-def)
  apply (simp-all add: DEF-X-up LESS-def div-def zero-def minint-def TRUE-def)
  apply (auto simp : quotient-sign-plus uminus-def minus-def one-def div-def)
done

lemma quotient-sign-minus1 :
  assumes 0: d ~= minint
  assumes 1: LESS d 0 = TRUE
  assumes 2: LESS 0 n = TRUE
  assumes 3: LE (− d) (n) = TRUE
  shows     LESS (n div d) 0 = TRUE
  apply (subgoal-tac DEF d & DEF n & DEF (−d))
  defer
  apply (insert 0 1 2 3)
  apply (drule DEF-isTRUE)
  apply (drule DEF-isTRUE)
  apply auto

  apply (rule DEF-isTRUE-I)
  apply (subgoal-tac d ≠ 0)
  apply (simp)
  prefer 2
 apply (auto simp: DEF-X-up LESS-def LE-def div-def zero-def TRUE-def uminus-def)
  apply (rule quotient-sign-minus1)
  apply (drule iffD1 [OF le-quasi-def])
  defer 1
  apply (drule iffD1 [OF le-quasi-def]) back
  defer 1
  apply (subst uminus-homom[symmetric])
  apply (simp-all add: JavaIntegersDef.zero-def minint-def
                    JavaIntegersRing.JavaInt-le-def)
  done
```

```
lemma quotient-sign-minus2 :
  assumes 1: LESS 0 d = TRUE
  assumes 2: LESS n 0 = TRUE
  assumes 3: n ≠ minint
  assumes 4: LE n (− d) = TRUE
  shows     LESS (n div d) 0 = TRUE
  apply (subgoal-tac DEF d & DEF n)
  defer
  apply (insert 1 2 3 4)
  apply (drule DEF-isTRUE)
  apply (drule DEF-isTRUE)
  apply auto
  apply (rule DEF-isTRUE-I)
  apply (subgoal-tac d ≠ 0)
  apply (simp)
  prefer 2
  apply (auto simp: DEF-X-up LESS-def div-def zero-def TRUE-def)
  apply (rule quotient-sign-minus2)
  apply (auto simp: minint-def NEG-def LE-def abs-def
                    TRUE-def NEG-def less-def zero-def
                    uminus-def one-def)
  apply (drule  iffD1 [OF le-quasi-def])
  defer 1
  apply (drule  iffD1 [OF le-quasi-def]) back
  defer 1
  apply (erule swap)
  apply (rule Rep-JavaInt-inject[THEN iffD1])
  defer 1
  apply (drule le-neg-bound)
  apply (simp-all add: JavaIntegersDef.zero-def MinInt-def)
  done


lemma neg-mod-bound:
  assumes 1: LESS b 0 = TRUE
  assumes 2: b ≠ minint
  assumes 3: DEF a
  shows     LESS b (a mod b) = TRUE
  apply (subgoal-tac DEF b)
  defer
  apply (insert 1 2 3)
  apply (drule DEF-isTRUE)
  apply auto
  apply (rule DEF-isTRUE-I)
  apply (subgoal-tac b ≠ 0)
  apply (simp)
  prefer 2
```

apply (*auto simp*: *DEF-X-up LESS-def mod-def zero-def TRUE-def*)
apply (*rule neg-mod-bound*)
apply (*auto simp*: *minint-def*)
done

lemma *pos-mod-bound*:
assumes *1*: *LESS 0 b* = *TRUE*
assumes *2*: *DEF a*
shows *LESS* (*a mod b*) *b* = *TRUE*
apply (*subgoal-tac DEF b*)
defer
apply (*insert 1 2*)
apply (*drule DEF-isTRUE*)
apply *auto*
apply (*rule DEF-isTRUE-I*)
apply (*subgoal-tac b* $\neq$ *0*)
apply (*simp*)
prefer *2*
apply (*auto simp*: *DEF-X-up LESS-def mod-def zero-def TRUE-def*)
apply (*rule pos-mod-bound*)
apply (*auto*)
done

lemma *neg-mod-sign*:
assumes *1*: *LESS a 0* = *TRUE*
assumes *2*: *b* $\neq$ *0*
assumes *3*: *DEF b*
shows *LE* (*a mod b*) *0* = *TRUE*
apply (*subgoal-tac DEF a*)
defer
apply (*insert 1 2 3*)
apply (*drule DEF-isTRUE*)
apply *auto*
apply (*rule DEF-isTRUE-I*)
apply (*subgoal-tac b* $\neq$ *0*)
apply (*simp*)
prefer *2*
apply (*auto simp*: *DEF-X-up LE-def LESS-def mod-def zero-def TRUE-def*)
apply (*rule neg-mod-sign*)
apply (*auto*)
done

lemma *pos-mod-sign*:
assumes *1*: *LE 0 a* = *TRUE*
assumes *2*: *b* $\neq$ *0*
assumes *3*: *b* $\neq$ *minint*
assumes *4*: *DEF b*

shows *LE 0* (*a mod b*) = *TRUE*
apply (*subgoal-tac DEF a*)
defer
apply (*insert 1 2 3 4*)
apply (*drule DEF-isTRUE*)
apply *auto*
apply (*rule DEF-isTRUE-I*)
apply (*subgoal-tac b* $\neq$ *0*)
apply (*simp*)
prefer *2*
apply (*auto simp*: *DEF-X-up LE-def LESS-def mod-def zero-def TRUE-def*)
apply (*rule pos-mod-sign*)
apply (*auto simp*: *minint-def*)
done

lemma *JavaInt-mod-less*:
assumes *1*: *b* $\neq$ *0*
assumes *2*: *b* $\neq$ *minint*
assumes *3*: *DEF a*
assumes *4*: *DEF b*
shows *LESS* (*abs* (*a mod b*)) (*abs b*) = *TRUE*
apply (*insert 1 2 3 4*)
apply (*rule DEF-isTRUE-I*)
apply (*simp*)
apply (*auto simp*: *DEF-X-up LE-def LESS-def mod-def zero-def TRUE-def abs-def*)
apply (*rule JavaInt-mod-less*)
apply (*auto simp*: *minint-def*)
done

end

19 Conclusions and Future Work

In this report we presented a formalization of Java's two's-complement integral types in Isabelle/HOL. Our formalization includes both normal and exceptional behavior (based on one exception). The normal behaviour Java integers presented here form a ring, and this even extends to the exceptional behaviour model with the only change that the inverse element rule must be weakend to $DEF(A) \Longrightarrow A + (-A) = 0$. Both for the normal and the exceptional behaviour model, a number of properties for the Java division and remainder operation have been proven (and tested against some implementations of the Java compiler). We get a two's-complement representation by redefining the conversion function *number_of* which is already provided for *int*. This representation can be used for those operators that are defined

bitwise, but this theory has only been developed to a very basic extent. Altogether, the existing Isabelle theories make it relatively easy to achieve standard number-theoretic properties for types that are defined as a subset of the Isabelle/HOL integers.

Such a formalization is a necessary prerequisite for the verification of efficient arithmetic Java programs such as encryption algorithms, in particular in tools like Jive [MPH00] that generate verification conditions over arithmetic formulae from such programs.

Our formalization of the normal behavior is based on a direct analysis of the Java Language Specification [GJSB00] and led to the discovery of several underspecifications and ambiguities (see 5 (4), 16.1, 15.1, 17). These underspecifications are highly undesirable since even compliant Java compilers may interpret the same program differently, leading to unportable code. In the future, we strongly suggest to supplement informal language definitions by machine-checked specifications like the one we present in this report as a part of the normative basis of a programming language.

We applied the technique of mechanized theory morphisms (developed in [BW03]) to our Java arithmetic model in order to achieve a clear separation of concerns between normal and exceptional behavior. Moreover, we showed that the concrete exceptional model can be exchanged — while controlling the exact side-conditions that are imposed by a concrete exceptional model. For the future, this leaves the option to use a lifting to the *exception state monad* [LHJ95] mapping the type JAVAINT to state $\Rightarrow$($\mathsf{JavaInt}_\perp$,state) in order to give semantics to expressions with side-effects like `i++ + i`.

Of course, more rules can be added to our theory in order to allow effective automatic computing of large (ground) expressions — this has not been in the focus of our interest so far. With respect to proof automation in JavaInt, it is an interesting question whether arithmetic decision procedures of most recent Isabelle versions (based on Cooper's algorithm for Presburger Arithmetic) can be used to decide analogous formulae based on machine arithmetic. While an *adoption* of these procedures to Java arithmetic seems impossible (this would require cancellation rules such as $(a \leq b) = (k \times a \leq k \times b)$ for nonnegative k which do not hold in Java), it is possible to retranslate JavaInt formulae to standard integer formulae; remainder subexpressions can be replaced via $P(a \bmod b) = \exists m.\ 0 \leq m < a \wedge (a - m) \mid b \wedge P(m)$, such that finally a Presburger formula results. Since a translation leads to an exponential blow-up in the number of quantifiers (a critical feature for Cooper's algorithm), it remains to be investigated to what extent this approach is feasible in practice.

References

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AS95] Mark D. Aagaard and Carl-Johan H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *International Conference on Computer Aided Design (ICCAD)*, pages 7–10. IEEE Computer Society, Nov 1995.

[BJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer Verlag, 2001.

[Bon] Didier Bondyfalat. Long integer division in Coq (algorithm divide and conquer). http://www-sop.inria.fr/lemme/Didier.Bondyfalat/DIV/.

[BRS+00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[BS02] Bernhard Beckert and Steffen Schlager. Integer arithmetic in the specification and verification of Java programs. In *Proceedings, Workshop on Tools for System Design and Verification (FM-TOOLS), Reisensburg, Germany*, pages 7–14, 2002.

[BW03] Achim D. Brucker and Burkhart Wolff. Using theory morphisms for implementing formal methods tools. In Herman Geuvers and Freek Wiedijk, editors, *Types 2002, Proceedings of the workshop Types for Proof and Programs*, Lecture Notes in Computer Science, Nijmegen, 2003. Springer Verlag.

[CnM95] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Higher Order Logic Theorem Proving and its Applications*, 1995.

[Fox01] A. C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge Computer Laboratory, April 2001.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, June 2000.

[GM93a] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[GM93b] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[Gol02] D. Goldberg. Computer arithmetic. In J.L. Hennessy and D.A. Patterson, editors, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2002.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer Verlag.

[HD99] Stefan Höreth and Rolf Drechsler. Formal verification of word-level specifications. In *IEEE Design, Automation and Test in Europe (DATE)*, Munich, 1999.

[Her03] Marc Herbstritt. E-mail communication, May 2003. Chair of Computer Architecture, Uni Freiburg.

[Jac03] Bart Jacobs. Java's integral types in PVS. *Submitted*, 2003.

[Jav] Java API Specification. http://java.sun.com/j2se/1.4.1/docs/api/java/.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors,

TACAS00, Tools and Algorithms for the Construction and Analysis of Systems, volume 276 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2000.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, New York, NY, USA, 1994.

[Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Technische Universität München, 1998.

[SBW02] Christoph Scholl, Bernd Becker, and Thomas Weis. On WLCDs and the complexity of word-level decision diagrams — a lower bound for division. *Formal Methods in System Design*, 20(3):311–326, 2002.

[ST99] Donald Sannella and Andrzej Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*, 31(3es), 1999.

[Sun00] Sun Microsystems, Inc., Palo Alto, CA. *Java Card^{TM} 2.1.1 Specifications – Release Notes*, May 2000.

[The02] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V7.3*, 2002.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. MIT Press, 1993.