Java's Integral Types in PVS

Bart Jacobs

`bart@cs.kun.nl`
`www.cs.kun.nl/~bart` `www.verificard.org`.

Dep. Computer Science, Univ. Nijmegen, NL

Contents

I. Example programs

Program 1

```
int program1 () {
  for (byte b = Byte.MIN_VALUE;
            b <= Byte.MAX_VALUE; b++) {
    if (b == 0x90) { return 1000; }
  }
  return 100;
}
```

This method will hang, because:

- loop condition never fails, since increment wraps around
- byte `b` $\in [-128, 127]$ never reaches integer value `0x90` = 144.

Program 2

```
int program2 () {
  int n = 0;
  while (-1 << n != 0) { n++; }
  return n;
}
```

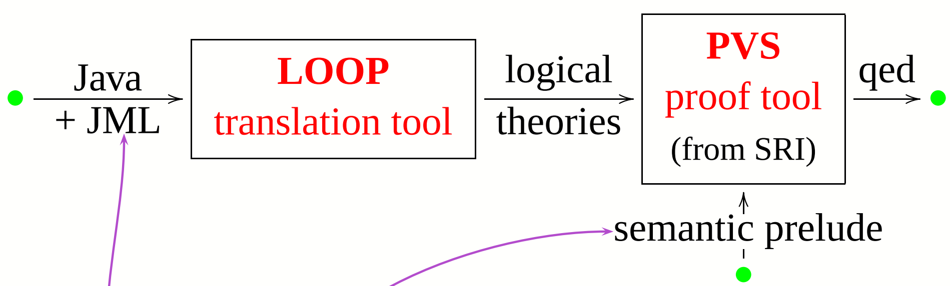
This method will also hang, because:

- -1 = `0xFFFFFFFF` needs 32 left-shifts to become 0
- Java uses only the five lower-order bits of `n` in `-1 << n`, which can be at most $2^5 - 1 = 31$.

Program 3, with JML annotation

```
/*@
  @ normal_behavior
  @ requires true;
  @ assignable \nothing;
  @ ensures \result ==
  @   (short)((b >= 0) ? b :  (b + 256));
  @*/
private short sh(byte b) {
  return (short)(b & 0xFF);
}
```


LOOP project: overview



- JML annotations become PVS predicates, which should be proved for the (translated) Java code.
- The semantic prelude contains the semantics in PVS of Java language constructs like composition, if-then-else, while, try-catch-finally, . . .

LOOP characteristics & results

- Translation covers essentially all of sequential Java and core of JML.
- Shallow embedding: Java methods become PVS functions.
- Program logics are proven sound in PVS, and applied within PVS
- Recent major case study (100s lines of code):
  - commercial, already tested smart card applet
  - possible exception detected
  - bug, but no security compromise

Observations

- Proper understanding of integral types is necessary for correct programming
- This is a non-entirely-trivial matter
- Also security risk involved, e.g. in security protocol:
  - `short seq` is sequence number, incremented with every run
  - overflow enables replay attack
- IEEE standard exists for floats, but not for integrals
- In program verification integral bounds are traditionally ignored.

No longer acceptable! Formalisation is needed.

Relevance for smart cards

- Smart cards have limited (memory) resources. Thus, programmers choose integral types as small as possible, and ***over/underflow*** is likely.
- Communication uses byte sequences (APDU's) and bit level operations to extract parameters & data.
- Marlet & Métayer (Trusted Logic): unwanted overflow must be avoided:

  bad: **`if (balance + credit > maxBalance) ..`**

  good: **`if (balance > maxBalance - credit) ..`**

  (where **`credit <= maxBalance`** is invariant)

II. Integral types in Java

Java's primitive types

- Recall that Java's primitive types are:

  `byte short int long char`
  `float double boolean`

- The first five of these describe the ***integral types***:

`byte`	8 bits, signed
`short`	16 bits, signed
`int`	32 bits, signed
`long`	64 bits, signed
`char`	16 bits, unsigned (for unicode characters)

- Only the **`byte`**, **`short`** are relevant in Java Card (and **`int`** in more recent cards).

Java's bounded arithmetic

- In Java:

$$\texttt{minint} = \texttt{0x80000000} = -2^{31}$$
$$\texttt{maxint} = \texttt{0x7FFFFFFF} = 2^{31}-1$$

- They satisfy for instance:

```
      minint - 1 == maxint
      maxint + 1 == minint
     minint * -1 == minint
 maxint * maxint == 1
     minint / -1 == minint
```


How to formalise?

- Via bounded intervals of integers, such as $\mathsf{int} = [-2^{31}, 2^{31}-1] \subseteq \mathbb{Z}$.
  - Carried out by Rauch & Wolff in Isabelle
  - Relies on difficult definitions (division, bitwise-and)
  - So far only for Java's `int`; not integrated in Jive verification environment
- Via bit vectors $b_1 \ldots b_\ell$, of length $\ell = 8, 16, 32, 64$.
  - Current approach, building on basic PVS library.
  - Low level definitions, yielding more abstract results
  - Integrated in Loop tool & used in several verifications

III. PVS's bitvector library

PVS 2.0 bitvector library I

- Basics developed by SRI, NASA, Rockwell, mainly for hardware verification.
- Bitvector length is parameter N:
```
    bvec[N] = [below(N) -> bit]
```
  where
```
    below(N) = {0,1,..,N-1}
    bit = {0,1}
```
- Unsigned interpretation:
  `bv2nat :  [bvec[N]->{0,1,..,2`^{N}`-1}]`
- Signed interpretation:
  `bv2int :  [bvec[N]->{-2`$^{N-1}$`,..,2`$^{N-1}$`-1}]`

  (Both functions are bijective)

PVS 2.0 bitvector library II

Typical result, with over- and under-flow:

$$\mathsf{bv2int}(a+b) = \begin{cases} \mathsf{bv2int}(a) + \mathsf{bv2int}(b) & \\ \quad \text{if } -2^{N-1} \leq \mathsf{bv2int}(a) + \mathsf{bv2int}(b) & \\ \quad \text{and } \mathsf{bv2int}(a) + \mathsf{bv2int}(b) < 2^{N-1} & \\ \mathsf{bv2int}(a) + \mathsf{bv2int}(b) - 2^N & \\ \quad \text{if } \mathsf{bv2int}(a) \geq 0 \text{ and } \mathsf{bv2int}(b) \geq 0 & \\ \mathsf{bv2int}(a) + \mathsf{bv2int}(b) + 2^N & \\ \quad \text{otherwise.} & \end{cases}$$

PVS 2.0 bitvector library III

- Basic definitions are given: +, -, shift, bitwise ops, etc.
- Multiplication, division, remainder are missing, but needed for Java.
- Also no widening & narrowing to move for instance between `byte` and `short`.

IV. Widening and Narrowing

Definitions

We seek functions:

$$\begin{tikzcd} \mathsf{bvec}(N) \arrow[r, bend left, "\mathsf{widen}"] & \mathsf{bvec}(2*N) \arrow[l, bend left, "\mathsf{narrow}"] \end{tikzcd}$$

They can be defined as:

$$\begin{aligned} \mathsf{widen}(a) &= \lambda i \colon \mathsf{below}(2*N).\ \begin{cases} a(i) & \text{if } i < N \\ a(N-1) & \text{else} \end{cases} \\ \mathsf{narrow}(A) &= \lambda i \colon \mathsf{below}(N).\ A(i) \end{aligned}$$

Results

$$\mathsf{bv2int}(\mathsf{widen}(a)) \;=\; \mathsf{bv2int}(a)$$

$$\begin{aligned}\mathsf{bv2int}\big(\mathsf{widen}(a)+\mathsf{widen}(b)\big) &= \mathsf{bv2int}(a)+\mathsf{bv2int}(b)\\ \mathsf{bv2int}\big(-\mathsf{widen}(a)\big) &= -\mathsf{bv2int}(a).\end{aligned}$$

General theme: after widening no overflow

$$\mathsf{narrow}(\mathsf{widen}(a)) \;=\; a$$

$$\begin{aligned}\mathsf{narrow}\big(A+B\big) &= \mathsf{narrow}(A)+\mathsf{narrow}(B)\\ \mathsf{narrow}\big(-A\big) &= -\mathsf{narrow}(A).\end{aligned}$$

Example

For **`byte b`**, **`short s`**, a Java expression

```
(short)(b + 2*s)
```

is translated into PVS as:

$$\mathsf{narrow}\big(\mathsf{widen}(\mathsf{widen}(b)) + 2 * \mathsf{widen}(s)\big)$$

because the arguments are "promoted" in Java to 32 bit integers before addition and multiplication are applied.

V. Multiplication for bitvectors

Idea

$$\begin{array}{cl} a_1 \ldots a_n & \\ \underline{b_1 \ldots b_n} \;\times & \\ a_1 \ldots a_n & \text{if } b_n = 1 \\ \vdots & \\ a_1 \ldots a_n 0 \ldots 0 & \text{if } b_i = 1 \\ \vdots & \\ \hline \text{multiplication result} \;+ & \end{array}$$

- Decide on least significant bit of right-shifted b's
- Add resulting left-shifted a's.
  (Actually, we left-shift the adds)

Implementation

Recursive definition:

$$a \ast b \;=\; \textsf{times-rec}(b, a, N)$$

where—using lsh = left-shift, rsh = right-shift,

$$\begin{array}{l}\textsf{times-rec}(b, a, n) \\ \quad = \begin{cases} \overrightarrow{0} \quad \text{if } n = 0 \\ a + \textsf{lsh}(\textsf{times-rec}(\textsf{rsh}(b), a, n-1)) \\ \qquad \text{if } n > 0 \text{ and } b(0) = 1 \\ \textsf{lsh}(\textsf{times-rec}(\textsf{rsh}(b), a, n-1)) \\ \qquad \text{if } n > 0 \text{ and } b(0) = 0 \end{cases}\end{array}$$

Results

- Definition amounts to iterated additions (with possible overflows).
- $(\textsf{bvec}(N), \ast, \mathbf{1})$ is a commutative monoid, and $\ast$ preserves the group structure $(\textsf{bvec}(N), +, \overrightarrow{0}, -)$.
- After widening no overflow:
  $\textsf{bv2int}(\textsf{widen}(a) \ast \textsf{widen}(b)) \;=\; \textsf{bv2int}(a) \ast \textsf{bv2int}(b)$
- Narrowing commutes with multiplication:
  $\textsf{narrow}(A \ast B) \;=\; \textsf{narrow}(A) \ast \textsf{narrow}(B)$

What the Java Language Spec says

- From the previous results:
  $a \ast b \;=\; \textsf{narrow}\big(\textsf{widen}(a) \ast \textsf{widen}(b)\big)$
- This is precisely what is in
  the Java Language Specification (2^{nd} ed, §§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.

VI. Division and Remainder

Definition

- Definition in two stages:
  - ***unsigned*** using pencil-and-paper approach, implemented as (standard) register-style machine algorithm
  - ***signed*** via (non-standard) case distinctions
- Non-trivial invariant is needed to prove correctness
- Uniqueness of division and remainder needed for reasoning

Division and remainder are strange

- Main property $(a/b) * b + (a\%b) = a$.
- Standard outcomes when a and b have equal signs:

$$\begin{array}{rclcrcl} 5 \;/\; 3 & = & 1 & \qquad & -5 \;/\; -3 & = & 1 \\ 5 \;\%\; 3 & = & 2 & & -5 \;\%\; -3 & = & -2 \end{array}$$

- But different signs are funny:

$$\begin{array}{rclcrcl} 5 \;/\; -3 & = & -1 & \qquad & -5 \;/\; 3 & = & -1 \\ 5 \;\%\; -3 & = & 2 & & -5 \;\%\; 3 & = & -2 \end{array}$$

General result (incomplete)

$$\begin{array}{l} \big(\mathsf{bv2int}(a) > 0 \;\&\; \mathsf{bv2int}(b) < 0\big) \text{ or} \\ \qquad\qquad \big(\mathsf{bv2int}(a) < 0 \;\&\; \mathsf{bv2int}(b) > 0\big) \\ \textbf{and not: } \exists n \in \mathbb{Z}.\ \mathsf{bv2int}(a) = n * \mathsf{bv2int}(b) \\ \textbf{implies} \\ \mathsf{bv2int}(a \;/\; b) \;=\; \mathsf{floor}\big(\mathsf{bv2int}(a) \,/\, \mathsf{bv2int}(b)\big) + 1 \end{array}$$

where $\mathsf{floor}(x) \leq x < \mathsf{floor}(x) + 1$

Such general formulations are results, not definitions

JLS properties hold

the quotient produced for operands n and d that are integers after binary numeric promotion is an integer value q whose magnitude is as large as possible while satisfying $|d * q| \leq |n|$; moreover, q is positive when and n and d have the same sign, but q is negative when and n and d have opposite signs. 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.

The remainder operation for operands that are integers after binary numeric promotion produces a result value such that $(a/b) * b + (a\%b)$ is equal to a. This identity holds even in the special case that the dividend is the

VII. Integral types in JML

JML assertions

- JML is becoming the standard specification language for Java, developed as open, community effort.
- Range of tools available, for type checking, run-time assertion checking, static analysis (ESC/Java), formal verification (Loop, Krakatoa, Jive, Jack)
- Big question: how should integral types be interpreted in assertions?
  - ***Current situation:*** bounded, like in Java.
  - ***Future:*** choice between bounded / unbounded / "safe", both for Java and for JML.
    [Work of Chalin & Kiniry]

Overflow in specification

overflow possible

```
/*@
  @ normal_behavior
  @ requires x >= 0;  && x <= 2147390966;
  @ assignable \nothing;
  @ ensures \result * \result <= x &&
  @   x < (\result+1) * (\result+1);
  @   && \result < 46340;
  @*/
int sqrt(int x) {
  int count = 0, sum = 1;
  while (sum <= x) {
    count++; sum += 2 * count + 1; }
  return count;
}
```


VIII. Conclusions

Conclusions

- We have given an extension of the PVS bitvector library for software verification.
- Hence much emphasis on widen/narrow properties
- Things that “everybody knows”, but hard to find and get right. Typical theorem prover work.
- Used in “advanced” program verification work at Nijmegen, esp. for smart cards
- Use in JML assertions not settled yet
- This extension of PVS 2.0 library is part of recently released PVS 3.0.