**Range of primitive integer types in Java**

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>$[-128, 127]$</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>$[-32768, 32767]$</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>$[-2^{31}, 2^{31} - 1]$</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>$[-2^{63}, 2^{63} - 1]$</td>
<td>64</td>
</tr>
</tbody>
</table>
Examples

Valid for Java integer semantics

\[
\text{MAX\_INT} + 1 \div \text{MIN\_INT}
\]

\[
\text{MIN\_INT} \times (-1) \div \text{MIN\_INT}
\]

\[
\exists x, y : \text{int}. x \neq 0 \land y \neq 0 \land x \times y \div 0
\]
Examples

Valid for Java integer semantics

\[
\text{MAX\_INT} + 1 \div \text{MIN\_INT}
\]

\[
\text{MIN\_INT} \times (-1) \div \text{MIN\_INT}
\]

\[
\exists x, y: \text{int}. x \neq 0 \land y \neq 0 \land x \times y \div 0
\]

Not valid for Java integer semantics

\[
\forall x: \text{int}. \exists y: \text{int}. y > x
\]
Examples

Valid for Java integer semantics

\[
\text{MAX\_INT} + 1 \div \text{MIN\_INT}
\]

\[
\text{MIN\_INT} \times (-1) \div \text{MIN\_INT}
\]

\[
\exists x, y : \text{int} . x \neq 0 \land y \neq 0 \land x \times y \div 0
\]

Not valid for Java integer semantics

\[
\forall x : \text{int} . \exists y : \text{int} . y > x
\]

Not a sound rewrite rules for Java integer semantics

\[
x + 1 > y + 1 \leadsto x > y
\]
General Problem revisited

- semantic gap between $\mathbb{Z}$ and Java integers
- defining a JavaDL semantics for Java integers that...
  - is a correct data refinement of $\mathbb{Z}$
  - reflects Java integer semantics

Req. 1
Req. 2
General Problem revisited

- semantic gap between \( \mathbb{Z} \) and Java integers
- defining a JavaDL semantics for Java integers that...
  - is a correct data refinement of \( \mathbb{Z} \)  
  - reflects Java integer semantics

3 approaches

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Description</th>
<th>Req. 1</th>
<th>Req. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{OCL} )</td>
<td>corresponds to semantics of ( \mathbb{Z} )</td>
<td>( \checkmark )</td>
<td>( X )</td>
</tr>
<tr>
<td>( S_{Java} )</td>
<td>corresponds to Java semantics</td>
<td>( X )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>( S_{KeY} )</td>
<td>combination of ( S_{OCL} ) and ( S_{Java} )</td>
<td>( \checkmark )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

Req. 1

Req. 2
Semantics $S_{OCL}$

$S_{OCL}$ assigns Java integers semantics of $\mathbb{Z}$

- Req. 1 trivially fulfilled
- Req. 2 violated, incorrect programs can be “verified”

Example:

\[ \models_{S_{OCL}} \forall x : \text{int}. (y = x + 1; ) y = x +_\mathbb{Z} 1 \]

**but for** $x = \text{MAX\_INT}$ program not correct
$S_{\text{Java}}$ assigns Java integers the semantics defined in Java Language Specification

- Req. 1 violated
  - several abstract states mapped onto one concrete state
- Req. 2 trivially fulfilled

No incorrect programs can be verified, but violation of Req. 1 leads to “incidentally” correct programs!
Our Approach

Approach

- **types** byte, short, int, long have semantics $S_{Java}$
- **additional virtual types** arithByte, arithShort, arithInt, and arithLong (called “arithmetical types”) with following semantics:
Our Approach

Approach

- **types** `byte, short, int, long` have semantics $S_{Java}$
- **additional virtual types** `arithByte, arithShort, arithInt, and arithLong` (called “arithmetical types”) with following semantics:
  - arithmetical types have infinite range
  - operators are **underspecified**:
    - Semantics as in $\mathbb{Z}$ but semantics unspecified if both arguments are in valid range but result is not.
Our Approach

Approach

- **types** `byte`, `short`, `int`, `long` have semantics $S_{Java}$

- **additional virtual types** `arithByte`, `arithShort`, `arithInt`, and `arithLong` (called “arithmetical types”) with following semantics:
  - arithmetical types have infinite range
  - operators are underspecified:
    - Semantics as in $\mathbb{Z}$ but semantics unspecified if both arguments are in valid range but result is not.
Semantics $S_{KEY}$ of $\text{arithByte, arithShort, arithInt, arithLong}$

Range: infinite ($\mathbb{Z}$)

Operations, e.g. $+$ on $\text{arithInt}$:

$+: \text{arithInt} \times \text{arithInt} \rightarrow \text{arithInt}$

3 cases:
Semantics $S_{KeY}$ of arithByte, arithShort, arithInt, arithLong

Range: infinite ($\mathbb{Z}$)

Operations, e.g. $+$ on arithInt:

$+: \text{arithInt} \times \text{arithInt} \rightarrow \text{arithInt}$

3 cases:

- both args. and result are in valid range (e.g. $2+3 \div 5$)

  “normal” case
Semantics $S_{KeY}$ of arithByte, arithShort, arithInt, arithLong

Range: infinite ($\mathbb{Z}$)

Operations, e.g. $+$ on arithInt:

$+: \text{arithInt} \times \text{arithInt} \rightarrow \text{arithInt}$

3 cases:

- both args. and result are in valid range (e.g. $2+3 \div 5$)  
  “normal” case

- both args. are in valid range but result is not (e.g. $\text{MAX}_{-}\text{INT}+1 \div ?$)  
  overflow case, not specified
Semantics $S_{KeY}$

Semantics $S_{KeY}$ of arithByte, arithShort, arithInt, arithLong

Range: infinite ($\mathbb{Z}$)

Operations, e.g. $+$ on arithInt:

$+ : \text{arithInt} \times \text{arithInt} \rightarrow \text{arithInt}$

3 cases:

- both args. and result are in valid range (e.g. $2+3 \div 5$)
  
  "normal" case

- both args. are in valid range but result is not (e.g. $\text{MAX\_INT} + 1 \div ?$)

  overflow case, not specified

- an arg. is not in valid range (e.g. $(\text{MAX\_INT} + _\mathbb{Z} 1) + 1 \div \text{MAX\_INT} + _\mathbb{Z} 2$)

  cannot happen during execution, only in logic
Reason for Underspecification

property provable in our calculus  \implies 

property independent of actual implementation of overflow case
Reason for Underspecification

property provable in our calculus \( \implies \)

property independent of actual implementation of overflow case

Main theorem

If

(i) \( \models_{s_{KeY}} \Gamma \rightarrow \langle p \rangle \psi \)

(ii) \( s \models_{s_{KeY}} \Gamma \)

(iii) \( s \) is a real state (i.e. all arith. variables in valid range)

then

(a) no overflow occurs in \( p' \) when started in \( s' \)

(b) \( p' \) terminates in state where property \( \psi \) holds.
A Sequent Calculus

Generation of pre-conditions that no overflow occurs built into calculus rules

Example: Rule for multiplication on arithmetical types

define predicate \( in_T(\cdot): \quad in_T(x) \iff \text{MIN}_T \leq x \leq \text{MAX}_T \)

1. \( \Gamma, \, in_{T_1}(x) \land in_{T_2}(y) \to in_T(x \times y) \vdash \{z \leftarrow x \times y\} \langle \phi \)

2. \( \Gamma, \, in_{T_1}(x), \, in_{T_2}(y), \, \neg in_T(x \times y) \vdash \langle z = \text{overflow}(x, y, "*" ); \rangle \phi \)

\( \Gamma \vdash \langle z = x*y; \rangle \phi \)
Software development following our approach:

- **Specification**: use of OCL type `INTEGER`
- **Implementation**: use of arithmetical types (e.g. `arithInt`)
- **Verification**: if all proof obligations are provable in our calculus
  - specified properties hold
  - no overflow occurs
- arithmetical types can safely be replaced by corresponding Java types