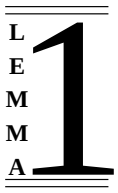


Lemma 1 Ltd.
27, Brook St.
Twyford
Berks
RG10 9NX

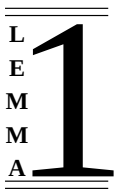
SMT-LIB in Z

Abstract

This document specifies models of the theories of SMT-LIB in the Z notation.

Version: 56bcb6f
Date: 15 May 2016
Reference: LEMMA1/ZSMTLIB/01
Pages: 21

Prepared by: R.D. Arthan
Tel: +44 7947 030 682
E-Mail: rda@lemma-one.com



0.1 Contents

σ

0.2 References

[1] Definitions of the SMT-LIB Logics. `http://smtlib.cs.uiowa.edu/logics.shtml`.

[2] Definitions of the SMT-LIB Theories. `http://smtlib.cs.uiowa.edu/theories.shtml`.

[3] Clark Barrett, Leonardo Mendonça de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB Initiative and the Rise of SMT - (HVC 2010 Award Talk). In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, volume 6504 of *Lecture Notes in Computer Science*, page 3. Springer, 2010.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[5] International Standards Organisation. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics.* ISO/IEC 13568:2002.

[6] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, 1992.

0.3 Changes History

2016/04/16 Initial draft.

2016/05/13 First commit to the ProofPower contrib repository.

0.4 Distribution

Rob Arthan	Lemma 1
Colin O'Halloran	D-RisQ
ProofPower Repo	`https://github.com/RobArthan/pp-contrib`

1 INTRODUCTION

This document gives a model written in the Z notation of each of the theories defined by the SMT-LIB standard version 2.5 [4, 2, 1]. This model was developed by Lemma 1 primarily to assist D-RisQ in reasoning about programming language datatypes that are modelled in Z and mapped into SMT-LIB to check assertions using and SMT solver. The model is made freely available to the Formal Methods community and comments are invited.

See [3] for an overview of SMT-LIB and its aims. See [6] for details of the Z notation, but note that the specification in this document makes use of features introduced in the ISO Z standard [5], specifically, it uses schemas with empty declaration parts to represent booleans as first-order values.

This document has been prepared using the document preparation tools supplied with ProofPower. It includes the ML commands required to type-check the Z and load it into a ProofPower database. Some small experimental proofs have been carried out and are included in the source of this document, but are omitted from the typeset document.

Please note this is work in progress.

2 THE SMT-LIB THEORIES

First we give the ML commands that set up the theory hierarchy for the SMT-LIB theories. Theory names comprise the SMT-LIB theory name prefixed with *ZS_*. The Core theory uses the Z Library up to and including the material on sequences. Sequences are used to represent functions that take two or more arguments.

The remaining theories depend on the Core theory and (in the case of *Reals_Ints* other SMT-LIB theories). The ML coommands to set up the theories are given in the following table.

Theory	ML set-up commands
Core	*open_theory*"*z_library*"; *new_theory*"*ZS_Core*";
Integer Numbers	*open_theory*"*ZS_Core*"; *new_theory*"*ZS_Ints*"; *new_parent*"*z_numbers*";
Real Numbers	*open_theory*"*ZS_Core*"; *new_theory*"*ZS_Reals*"; *new_parent*"*z_reals*";
Real and Integer Numbers	*open_theory*"*ZS_Core*"; *new_theory*"*ZS_Reals_Ints*"; *new_parent*"*ZS_Reals*"; *new_parent*"*ZS_Ints*";
Arrays	*open_theory*"*ZS_Core*"; *new_theory*"*ZS_ArraysEx*";
Bit Vectors	*open_theory*"*ZS_Core*"; *new_theory*"*ZS_FixedSizeBitVectors*";

2.1 Core

SML
```
open_theory"ZS_Core";
```

Standard Z is a form of first-order typed set theory. While the ProofPower system that we use to type-check and reason about Z actually supports a higher-order extension of Z, we prefer to stay within Standard Z in this specification. We will use what turns out to be a very convenient way of representing truth values as first-order objects following an idea of Sam Valentine. This uses a special case of the labelled record types provided in Z and referred to as schema types. The elements of schema types are called bindings. Z provides a convenient notation for denoting arbitrary subsets of schema types. For example, $[x,\ y\ :\ \mathbb{Z} \mid x\ <\ y]$ denotes the set of all bindings $(x == a, y == b)$ where $a < b$. In this notation, the declarations to the right of the vertical bar are called the signature. As a special case, $[|true]$ denotes the total set of the schema type over the empty signature, which has just one element. The power set of this type then has two elements, which we use to represent the two truth values.

z

$$\boldsymbol{Bool} == \mathbb{P}\ [|true]$$

z

$$\boldsymbol{True} == [|true]$$

z

$$\boldsymbol{False} == [|false]$$

Z has built-in operators (referred to collectively as the schema calculus) that combine sets of bindings by combining the defining properties of the sets with logical connectives. This gives us a direct representation of the propositional connectives on our chosen representation of truth values.

z

$$\boldsymbol{Not} : Bool \to Bool;$$
$$\boldsymbol{And}, \boldsymbol{Or}, \boldsymbol{Xor}, \boldsymbol{Implies} : Bool \to Bool \to Bool$$

$$\forall p : Bool \bullet Not\ p = (\neg_s\ p);$$
$$\forall p, q : Bool \bullet And\ p\ q = (p \wedge_s q);$$
$$\forall p, q : Bool \bullet Or\ p\ q = (p \vee_s q);$$
$$\forall p, q : Bool \bullet Xor\ p\ q = (\neg_s(p \Leftrightarrow_s q));$$
$$\forall p, q : Bool \bullet Implies\ p\ q = (p \Rightarrow_s q)$$

It is now very straightforward to define the chainable equality and pairwise distict predicates:

z

$$[A]$$

$$\boldsymbol{Equal}, \boldsymbol{Distinct} : seq\ A \to Bool$$

$$\forall s : seq\ A \bullet Equal\ s = [|\ \forall i, j : dom\ s \bullet s\ i = s\ j];$$
$$\forall s : seq\ A \bullet Distinct\ s = [|\ \forall i, j : dom\ s \bullet s\ i = s\ j \Leftrightarrow i = j]$$

We define if-then-else by pattern-matching:

z

$$[A]$$

$$\boldsymbol{ITE} : Bool \to A \to A \to A$$

$$\forall x, y : A \bullet ITE\ True\ x\ y = x;$$
$$\forall x, y : A \bullet ITE\ False\ x\ y = y$$

2.2 Integer Numbers

The integers from the Z toolkit provide the representation for the theory of Integers.

SML
```
open_theory "ZS_Ints";
```

Z
$$\boldsymbol{Int} == \mathbb{Z}$$

Note that (unlike Lisp) addition and multiplication are binary operators not operators on lists. Division and modulus are defined the same way in the ProofPower Z toolkit as they are in SMT-LIB, but Standard Z offers a different definition

Z
$$\boldsymbol{INeg}, \boldsymbol{IAbs} : Int \to Int;$$
$$\boldsymbol{ISub}, \boldsymbol{IAdd}, \boldsymbol{IMul}, \boldsymbol{IDiv}, \boldsymbol{IMod} : Int \to Int \to Int$$

$$\forall x : Int \bullet INeg\ x = \sim x;$$
$$\forall x : Int \bullet IAbs\ x = abs\ x;$$
$$\forall x, y : Int \bullet ISub\ x\ y = x - y;$$
$$\forall x, y : Int \bullet IAdd\ x\ y = x + y;$$
$$\forall x, y : Int \bullet IMul\ x\ y = x * y;$$
$$\forall x, y : Int \mid \neg\ y = 0 \bullet\ 0 \leq IMod\ x\ y < abs\ y;$$
$$\forall x, y : Int \mid \neg\ y = 0 \bullet\ x = (IDiv\ x\ y) * y + IMod\ x\ y$$

Z
$$\boldsymbol{ILE}, \boldsymbol{ILT}, \boldsymbol{IGE}, \boldsymbol{IGT} : seq\ Int \to Bool$$

$$\forall s : seq\ Int \bullet ILE\ s = [|\ \forall i, j : dom\ s \bullet\ i < j \Rightarrow s\ i \leq s\ j];$$
$$\forall s : seq\ Int \bullet ILT\ s = [|\ \forall i, j : dom\ s \bullet\ i < j \Rightarrow s\ i < s\ j];$$
$$\forall s : seq\ Int \bullet IGE\ s = [|\ \forall i, j : dom\ s \bullet\ i < j \Rightarrow s\ i \geq s\ j];$$
$$\forall s : seq\ Int \bullet IGT\ s = [|\ \forall i, j : dom\ s \bullet\ i < j \Rightarrow s\ i > s\ j]$$

Z
$$\boldsymbol{Divisible} : \mathbb{N}_1 \to Int \to Bool$$

$$\forall n, i : Int \bullet Divisible\ n\ i = [|\ \exists q{:}Int \bullet\ i = q{*}n\ ]$$

2.3 Reals

The ProofPower-Z library defines the real numbers as a separate type with the operators distinguished from those for the integers by a subscript.

SML

```
open_theory "ZS_Reals";
```

Z

$$\boldsymbol{Real} == \mathbb{R}$$

Z

$$\boldsymbol{RNeg} : Real \to Real;$$
$$\boldsymbol{RSub}, \boldsymbol{RAdd}, \boldsymbol{RMul}, \boldsymbol{RDiv} : Real \to Real \to Real$$

$$\forall x : Real \bullet RNeg\ x = \sim_R x;$$
$$\forall x, y : Real \bullet RSub\ x\ y = x -_R y;$$
$$\forall x, y : Real \bullet RAdd\ x\ y = x +_R y;$$
$$\forall x, y : Real \bullet RMul\ x\ y = x *_R y;$$
$$\forall x, y : Real \bullet RDiv\ x\ y = x /_R y$$

Z

$$\boldsymbol{RLE}, \boldsymbol{RLT}, \boldsymbol{RGE}, \boldsymbol{RGT} : seq\ Real \to Bool$$

$$\forall s : seq\ Real \bullet RLE\ s = [| \forall i, j : dom\ s \bullet i < j \Rightarrow s\ i \leq_R s\ j |];$$
$$\forall s : seq\ Real \bullet RLT\ s = [| \forall i, j : dom\ s \bullet i < j \Rightarrow s\ i <_R s\ j |];$$
$$\forall s : seq\ Real \bullet RGE\ s = [| \forall i, j : dom\ s \bullet i < j \Rightarrow s\ i \geq_R s\ j |];$$
$$\forall s : seq\ Real \bullet RGT\ s = [| \forall i, j : dom\ s \bullet i < j \Rightarrow s\ i >_R s\ j |]$$

2.4 Reals_Ints

SML

```
open_theory"ZS_Reals_Ints";
```

Z

$$\boldsymbol{ToReal} : Int \to Real$$

$$ToReal = real$$

z

$$\boldsymbol{ToInt} : Real \to Int$$

$$\forall\ x : Real \bullet\ let\ i == real\ (ToInt\ x) \bullet\ i \leq_R x <_R i +_R 1.0$$

z

$$\boldsymbol{IsInt} : Real \to Bool$$

$$\forall\ x : Real \bullet\ IsInt\ x = [|\ x \in ran\ ToReal\ |]$$

2.5 ArraysEx

SML

```
open_theory "ZS_ArraysEx";
```

z

$$Array[X,\ Y] == X \to Y$$

z

$$[X, Y]$$

$$\boldsymbol{Select} : Array[X,\ Y] \to X \to Y;$$
$$\boldsymbol{Store} : Array[X,\ Y] \to X \to Y \to Array[X,\ Y]$$

$$\forall a : Array[X,\ Y];\ x : X \bullet\ Select\ a\ x = a\ x;$$
$$\forall a : Array[X,\ Y];\ x : X;\ y : Y \bullet\ Store\ a\ x\ y = a \oplus \{x \mapsto y\}$$

2.6 FixedSizeBitVectors

SML

```
open_theory "ZS_FixedSizeBitVectors";
```

We represent SMTLIB bit vectors as (non-empty) sequences of bits:

z

$$Bit == \{0,\ 1\}$$

The bit vectors of lengths $1, 2, \ldots$ partition the set of all non-empty bit strings:

z

$$BitString == seq_1\ Bit$$

z

$$\boldsymbol{BitVec} : \mathbb{N}_1 \rightarrow \mathbb{P}\ BitString$$

$$\forall m : \mathbb{N}_1 \bullet\ BitVec\ m = \{b : BitString \mid \#b = m\}$$

Note that in SMTLIB, the length of a bit vector is always a statically known. So SMTLIB can and does impose static restrictions that we will have to model in Z by treating *BitVec* like a dependent subtype constructor.

We need some auxiliary functions to assist in defining the sequence operations. We need the function that converts a bit string into the natural number it represents (in binary, most-significant bit at the left):

z

$$\boldsymbol{BV2Nat} : BitString \rightarrow \mathbb{N}$$

$$\forall x : Bit \bullet\ BV2Nat\ \langle x \rangle = x;$$
$$\forall x : Bit;\ b : BitString \bullet\ BV2Nat\ (b \frown \langle x \rangle) = 2 * BV2Nat\ b + x$$

To define the conversion from natural numbers to bit vectors, we need natural number powers of integers. This may be defined already in later versions of ProofPower. The following ML defends against that:

SML

```
val _ = ⌜z(_^_)⌝ handle Fail _ => (
diag_line "Please ignore the above error message.";
```

z

$$fun\ 80\ rightassoc\ \_\ \widehat{}\ \_$$

z

$$\_\ \widehat{}\ \_ : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Z}$$

$$\forall i : \mathbb{Z} \bullet\ i\ \widehat{}\ 0 = 1;$$
$$\forall i : \mathbb{Z};\ j : \mathbb{N} \bullet\ i\ \widehat{}\ (j + 1) = i * i\ \widehat{}\ j$$

SML

```
mk_t (* conclude val ... handle Fail _ => ( ... *));
```

Now we can define the conversion from natural numbers to bit vectors of a given length. Note that the restriction $BitVec\ m \lhd BV2Nat$ is a bijection between $BitVec\ m$ and $0\ ..\ 2 \frown m - 1$ for any m.

z
$$\boldsymbol{Nat2BV} : \mathbb{N}_1 \to \mathbb{N} \to BitString$$
$$\forall m : \mathbb{N}_1;\ n : \mathbb{N} \bullet Nat2BV\ m\ n = ((BitVec\ m \lhd BV2Nat)^{\sim})\ (n\ mod\ 2 \frown m)$$

We need the logical operations on bits:

z
$$\boldsymbol{BitNot} : Bit \to Bit;$$
$$\boldsymbol{BitAnd}, \boldsymbol{BitOr} : Bit \to Bit \to Bit$$
$$\forall x : Bit \bullet BitNot\ x = 1 - x;$$
$$\forall x, y : Bit \bullet BitAnd\ x\ y = x * y;$$
$$\forall x, y : Bit \bullet BitOr\ x\ y = x + y - x{*}y$$

z
$$\boldsymbol{Concat} : BitString \to BitString \to BitString$$
$$\forall b1, b2 : BitString \bullet Concat\ b1\ b2 = b1 \frown b2$$

SMTLIB indexes bit vectors from right-to-left starting at 0. Z sequences are indexed left to right starting at 1. Hence in a bit vector of length n, SMTLIB index i corresponds to Z index $n - i$.

z
$$\boldsymbol{Extract} : \mathbb{N} \to \mathbb{N} \to BitString \nrightarrow BitString$$
$$\forall i, j : \mathbb{N} \bullet$$
$$dom\ (Extract\ i\ j) = \{b : BitString \mid \#b > i \geq j\};$$
$$\forall i, j : \mathbb{N} \bullet \forall b : dom\ (Extract\ i\ j) \bullet$$
$$Extract\ i\ j\ b = ((\#b - i)\ ..\ (\#b - j)) \upharpoonright b$$

For completeness, we define the binary bitwise logical operations on operands of mixed length to give a result that has the same length as the shorter operand. Mixed length operands are statically disallowed allowed in SMTLIB.

z

$$
\begin{array}{l}
\boldsymbol{BVNot} : BitString \to BitString; \\
\boldsymbol{BVAnd}, \boldsymbol{BVOr} : BitString \to BitString \to BitString \\
\hline
\forall s : BitString \bullet \\
\quad dom(BVNot\ s) = dom\ s \\
\land \quad (\forall i : dom\ s \bullet BVNot\ s\ i = BitNot(s\ i)); \\
\forall s, t : BitString \bullet \\
\quad dom(BVAnd\ s\ t) = dom\ s \cap dom\ t \\
\land \quad (\forall i : dom\ s \cap dom\ t \bullet BVAnd\ s\ t\ i = BitAnd\ (s\ i)\ (t\ i)); \\
\forall s, t : BitString \bullet \\
\quad dom(BVOr\ s\ t) = dom\ s \cap dom\ t \\
\land \quad (\forall i : dom\ s \cap dom\ t \bullet BVOr\ s\ t\ i = BitOr\ (s\ i)\ (t\ i))
\end{array}
$$

For the binary arithmetic operations on operands of mixed length we take the result to have the same length as the longer operand. Again, SMTLIB disallows mixed length operands statically. The treatment of zero divisors is just inhereited from the Z toolkit.

z

$$
\begin{array}{l}
\boldsymbol{BVNeg} : BitString \to BitString; \\
\boldsymbol{BVAdd}, \boldsymbol{BVMul}, \boldsymbol{BVUdiv}, \boldsymbol{BVUrem} : BitString \to BitString \to BitString \\
\hline
\forall s : BitString \bullet let\ m == \#s \bullet \\
\quad BVNeg\ s = Nat2BV\ m\ (2 \hat{}\ m - BV2Nat\ s); \\
\forall s, t : BitString \bullet let\ m == max\{\#s, \#t\} \bullet \\
\quad BVAdd\ s\ t = Nat2BV\ m\ (BV2Nat\ s + BV2Nat\ t); \\
\forall s, t : BitString \bullet let\ m == max\{\#s, \#t\} \bullet \\
\quad BVMul\ s\ t = Nat2BV\ m\ (BV2Nat\ s * BV2Nat\ t); \\
\forall s, t : BitString \bullet let\ m == max\{\#s, \#t\} \bullet \\
\quad BVUdiv\ s\ t = Nat2BV\ m\ (BV2Nat\ s\ div\ BV2Nat\ t); \\
\forall s, t : BitString \bullet let\ m == max\{\#s, \#t\} \bullet \\
\quad BVUrem\ s\ t = Nat2BV\ m\ (BV2Nat\ s\ mod\ BV2Nat\ t)
\end{array}
$$

For the shift operations on operands of mixed length we take the result to have the same length as the first operand. Again, SMTLIB disallows mixed length operands statically.

z

$$\boldsymbol{BVShl}, \boldsymbol{BVLshr} : BitString \to BitString \to BitString$$

$$\forall s, t : BitString \bullet let\ m == \#s \bullet$$
$$BVShl\ s\ t = Nat2BV\ m\ (BV2Nat\ s * 2\hat{}BV2Nat\ t);$$
$$\forall s, t : BitString \bullet let\ m == \#s \bullet$$
$$BVLshr\ s\ t = Nat2BV\ m\ (BV2Nat\ s\ div\ 2\hat{}BV2Nat\ t)$$

Finally we have the arithmetic comparison operator:

z

$$\boldsymbol{BVULT} : BitString \to BitString \to Bool$$

$$\forall s, t : BitString \bullet BVULT\ s\ t = [|\ BV2Nat\ s < BV2Nat\ t]$$

3 THE SMT-LIB LOGICS

In this section, we give Z models that support the restrictions and extensions to the various theories that are defined tn the SMT-LIB logics.

The ML commands to set up the theories for the logics are given in the following table.

Theory	ML set-up commands
Bit Vector Logics	*open_theory*"*ZS_FixedSizeBitVectors*"; *new_theory*"*ZS_BV_Extensions*";

3.1 Bit Vector Extensions

SML

```
open_theory "ZS_BV_Extensions";
```

Note that the family of new constans `(_ bvX n)` for numerals `X` and `n` introduced in the BV logic is implemented by the constant *Nat2BV* that is already defined in our model of the fixed length bit vector theory.

The four additional bitwise logical operations are defined in terms of basic ones just as in the SMT-LIB definition.

z

$$\boldsymbol{BVNand}, \boldsymbol{BVNor}, \boldsymbol{BVXor}, \boldsymbol{BVXnor} : BitString \to BitString \to BitString$$

$$\forall s, t : BitString \bullet BVNand\ s\ t = BVNot\ (BVAnd\ s\ t);$$
$$\forall s, t : BitString \bullet BVNor\ s\ t = BVNot\ (BVOr\ s\ t);$$
$$\forall s, t : BitString \bullet BVXor\ s\ t = BVOr\ (BVAnd\ s\ (BVNot\ t))\ (BVAnd\ (BVNot\ s)\ t);$$
$$\forall s, t : BitString \bullet BVXnor\ s\ t = BVOr\ (BVAnd\ s\ t)\ (BVAnd\ (BVNot\ s)\ (BVNot\ t))$$

We define the comparison operator in terms of equality rather than recursively. This extends the operation to operands of non-equal length so as to return `#b0`.

z

$$\boldsymbol{BVComp} : BitString \to BitString \to BitVec\ 1$$

$$\forall s, t : BitString \bullet BVComp\ s\ t = if\ s = t\ then\ \langle 1 \rangle\ else\ \langle 0 \rangle$$

Subtraction is defined in terms of addition and 2s complement negation just as in the SMT-LIB definition.

z

$$\boldsymbol{BVSub} : BitString \to BitString \to BitString$$

$$\forall s, t : BitString \bullet BVSub\ s\ t = BVAdd\ s\ (BVNeg\ t)$$

The definitions of the other signed arithmetic operators are a little clearer in Z if we extract the sign bit using sequence indexing rather than *Extract* and if we use Z if-then-else rather than *ITE*.

The SMT-LIB sdiv operator is 2s-complement signed division truncating towards 0. It is defined by cases on the signs of the operands using negation and unsigned division. The definition is equivalent to the following:

$$s \,\mathsf{sdiv}\, t = \mathsf{sgn}(s) \cdot \mathsf{sgn}(t) \cdot (\mathsf{abs}(s) \,\mathsf{udiv}\, \mathsf{abs}(t)).$$

In m-bit 2s-complement arithmetic, $s \,\mathsf{sdiv}\, t$ gives the correct signed result except *(i)* when $t = 0$, in which case the result is unspecified, and *(ii)* when $s = -2^{m-1}$ and $t = -1$, in which case the result is -2^{m-1} (i.e., the absolute value is correct, but the sign is wrong).

z

$$\boldsymbol{BVSdiv} : BitString \to BitString \to BitString$$

$$\begin{array}{l}
\forall s,\ t : BitString \bullet\ BVSdiv\ s\ t = \\
\quad (let\ msb_s == s\ 1;\ msb_t == t\ 1 \bullet \\
\qquad if \quad msb_s = 0 \land msb_t = 0 \\
\qquad then \quad BVUdiv\ s\ t \\
\qquad else\ if\ msb_s = 1 \land msb_t = 0 \\
\qquad then \quad BVNeg\ (BVUdiv\ (BVNeg\ s)\ t) \\
\qquad else\ if\ msb_s = 0 \land msb_t = 1 \\
\qquad then \quad BVNeg\ (BVUdiv\ s\ (BVNeg\ t)) \\
\qquad else \quad BVUdiv\ (BVNeg\ s)\ (BVNeg\ t))
\end{array}$$

The SMT-LIB srem operator is signed remainder with sign following the dividend. The definition is equivalent to the following:

$$s \,\mathsf{srem}\, t = \mathsf{sgn}(s) \cdot (\mathsf{abs}(s) \,\mathsf{urem}\, \mathsf{abs}(t)).$$

If $T = 0$ this is undefined, otherwise when t is an m-bit 2s-complement number, $0 \leq \mathsf{abs}(s) \,\mathsf{urem}\, \mathsf{abs}(t) < \mathsf{abs}(t) \leq 2^{m-1}$, hence the multiplication by $\mathsf{sgn}(s)$ will not overflow in m-bit arithmetic.

The operators sdiv and srem satisfy the following identities:

$$\begin{array}{rl}
\mathsf{abs}(s \,\mathsf{srem}\, t) \leq \mathsf{abs}(t) & \text{provided } t \neq 0 \\
s = (s \,\mathsf{sdiv}\, t) * t + s \,\mathsf{srem}\, t & \text{provided } t \neq 0 \text{ and } (s,t) \neq (-2^{m-1}, -1).
\end{array}$$

z

$$\boldsymbol{BVSrem} : BitString \to BitString \to BitString$$

$$\begin{array}{l}
\forall s,\ t : BitString \bullet\ BVSrem\ s\ t = \\
\quad (let \quad msb_s == s\ 1;\ msb_t == t\ 1 \bullet \\
\qquad if \quad msb_s = 0 \land msb_t = 0 \\
\qquad then \quad BVUrem\ s\ t \\
\qquad else\ if\ msb_s = 1 \land msb_t = 0 \\
\qquad then \quad BVNeg\ (BVUrem\ (BVNeg\ s)\ t) \\
\qquad else\ if\ msb_s = 0 \land msb_t = 1 \\
\qquad then \quad BVUrem\ s\ (BVNeg\ t) \\
\qquad else \quad BVNeg(BVUrem\ (BVNeg\ s)\ (BVNeg\ t)))
\end{array}$$

The SMT-LIB smod operator is signed remainder with sign following the divisor. The definition is equivalent to the following:

$$s \,\mathsf{smod}\, t = s - \lfloor s/t \rfloor \cdot t.$$

This is defined by cases in such a way as to give the correct result in 2s-complement arithmetic except when $t = 0$, in which case the result is undefined.

z

$$\boldsymbol{BVSmod} : BitString \to BitString \to BitString$$

$$\begin{array}{l}
\forall s,\ t : BitString \bullet\ BVSmod\ s\ t = \\
\quad (let\quad msb_s == s\ 1;\ msb_t == t\ 1 \bullet \\
\quad\ let\quad abs_s == if\ msb_s = 0\ then\ s\ else\ BVNeg\ s; \\
\qquad\quad abs_t == if\ msb_t = 0\ then\ t\ else\ BVNeg\ t \bullet \\
\quad\ let\quad u == BVUrem\ abs_s\ abs_t \bullet \\
\qquad\quad if\quad u = Nat2BV\ (\#s)\ 0 \\
\qquad\quad then\quad u \\
\qquad\quad else\ if\ msb_s = 0 \land msb_t = 0 \\
\qquad\quad then\quad u \\
\qquad\quad else\ if\ msb_s = 1 \land msb_t = 0 \\
\qquad\quad then\quad BVAdd\ (BVNeg\ u)\ t \\
\qquad\quad else\ if\ msb_s = 0 \land msb_t = 1 \\
\qquad\quad then\quad BVAdd\ u\ t \\
\qquad\quad else\quad BVNeg\ u)
\end{array}$$

Unsigned less-than is already defined in the fixed size bit vector theory. The bit vector extensions add the three unsigned comparison operators.

z

$$\boldsymbol{BVULE}, \boldsymbol{BVUGT}, \boldsymbol{BVUGE} : BitString \to BitString \to Bool$$

$$\begin{array}{l}
\forall s,\ t : BitString \bullet\ BVULE\ s\ t = [|\ BV2Nat\ s \leq BV2Nat\ t]; \\
\forall s,\ t : BitString \bullet\ BVULE\ s\ t = [|\ BV2Nat\ s > BV2Nat\ t]; \\
\forall s,\ t : BitString \bullet\ BVULE\ s\ t = [|\ BV2Nat\ s \geq BV2Nat\ t]
\end{array}$$

The BV logic extensions give the four signed comparison operators.

z

$$\boldsymbol{BVSLT}, \boldsymbol{BVSLE}, \boldsymbol{BVSGT}, \boldsymbol{BVSGE} : BitString \to BitString \to Bool$$

$$\forall s, t : BitString \bullet BVSLT\ s\ t = [|$$
$$s\ 1 = 1 \land t\ 1 = 0$$
$$\lor \quad s\ 1 = t\ 1 \land BV2Nat\ s < BV2Nat\ t];$$
$$\forall s, t : BitString \bullet BVSLT\ s\ t = [|$$
$$s\ 1 = 1 \land t\ 1 = 0$$
$$\lor \quad s\ 1 = t\ 1 \land BV2Nat\ s \leq BV2Nat\ t];$$
$$\forall s, t : BitString \bullet BVSGT\ s\ t = BVSLT\ t\ s;$$
$$\forall s, t : BitString \bullet BVSGE\ s\ t = BVSLE\ t\ s$$

The SMT-LIB operator ashr is an arithmetic right shift, i.e., it shifts in copies of the sign-bit from the left. (Note that left shifts are the same for signed and unsigned arithmetic.)

z

$$\boldsymbol{BVAshr} : BitString \to BitString \to BitString$$

$$\forall s, t : BitString \bullet BVAshr\ s\ t =$$
$$if\ s\ 1 = 0\ then\ BVLshr\ s\ t\ else\ BVNot\ (BVLshr\ (BVNot\ s)\ t)$$

The repeat operator replicates a bit vector a specified number of times. We define this using indexing rather than recursion.

z

$$\boldsymbol{BVRepeat} : \mathbb{N} \to BitString \to BitString$$

$$\forall j : \mathbb{N}; t : BitString \bullet$$
$$\#(BVRepeat\ j\ t) = j * \#t$$
$$\land \quad (\forall i : 1 \mathinner{..} j * \#t \bullet BVRepeat\ j\ t\ i = t\ ((i - 1)\ mod\ j) + 1)$$

The zero extension operator pads a bit vector on the left with a specified number of zeroes.

z

$$\boldsymbol{BVZeroExtend} : \mathbb{N} \to BitString \to BitString$$

$$\forall i : \mathbb{N}; t : BitString \bullet BVZeroExtend\ i\ t = ((1 \mathinner{..} i) \times \{0\}) \frown t$$

z

$$\boldsymbol{BVRotateLeft}, \boldsymbol{BVRotateRight} : \mathbb{N} \to BitString \to BitString$$

$$\forall i : \mathbb{N};\ t : BitString \bullet$$
$$BVRotateLeft\ i\ t = (((i + 1) \mathinner{..} \#t) \upharpoonright t) \frown ((1 \mathinner{..} i) \upharpoonright t);$$
$$\forall i : \mathbb{N};\ t : BitString \bullet$$
$$BVRotateRight\ i\ t = (((\#t - i + 1) \mathinner{..} \#t) \upharpoonright t) \frown ((1 \mathinner{..} (\#t - i)) \upharpoonright t)$$

4 INDEX

A SOME PROOFS

SML

```
val _ = open_theory "ZS_Core";
val _ = set_pc "z_language_ext";
val zs_bool_def : THM = z_get_spec⌜Bool⌝;
val zs_true_def : THM = z_get_spec⌜True⌝;
val zs_false_def : THM = z_get_spec⌜False⌝;

val zs_bool_u_thm = save_thm("zs_bool_u_thm", (
set_goal([], ⌜Bool = 𝕌⌝);
a(rewrite_tac[zs_bool_def]);
pop_thm()
));

val zs_bool_true_false_thm = save_thm("zs_u_true_false_thm", (
set_goal([], ⌜𝕌 = {true, false}⌝);
a(rewrite_tac[zs_true_def, zs_false_def] THEN REPEAT strip_tac THEN taut_tac);
pop_thm()
));

val zs_true_neq_false_thm = save_thm("zs_true_neq_false_thm", (
set_goal([], ⌜¬true = false⌝);
a(rewrite_tac[zs_true_def, zs_false_def]);
pop_thm()
));

val _ = open_theory "ZS_Ints";
val _ = set_pc "z_library";
val zs_int_def = z_get_spec ⌜Int⌝;
val zs_i_ops_def = z_get_spec ⌜INeg⌝;



local
        val thms = (strip_∧_rule o rewrite_rule [zs_int_def, z_get_spec⌜ℤ⌝]) zs_i_ops_def;
in
        val zs_mod_thm = (hd o tl o rev) thms;
        val zs_div_mod_thm = (hd o rev) thms;
end;

val zs_div_mod_thm = save_thm("zs_div_mod_thm", (
set_goal([], ⌜∀x, y : ℤ | ¬y = 0 • IDiv x y = x div y ∧ IMod x y = x mod y⌝);
a(REPEAT_UNTIL is_∧ strip_tac);
a(ante_tac (z_∀_elim⌜(i ≙ x, j ≙ y, d ≙ IDiv x y, r ≙ IMod x y)⌝ z_div_mod_unique_thm));
a(asm_rewrite_tac[]);
a(⇒_T (rewrite_thm_tac o eq_sym_rule));
a(ALL_FC_T (conv_tac o LEFT_C o LEFT_C o once_rewrite_conv)[zs_div_mod_thm]);
a(rewrite_tac[]);
```