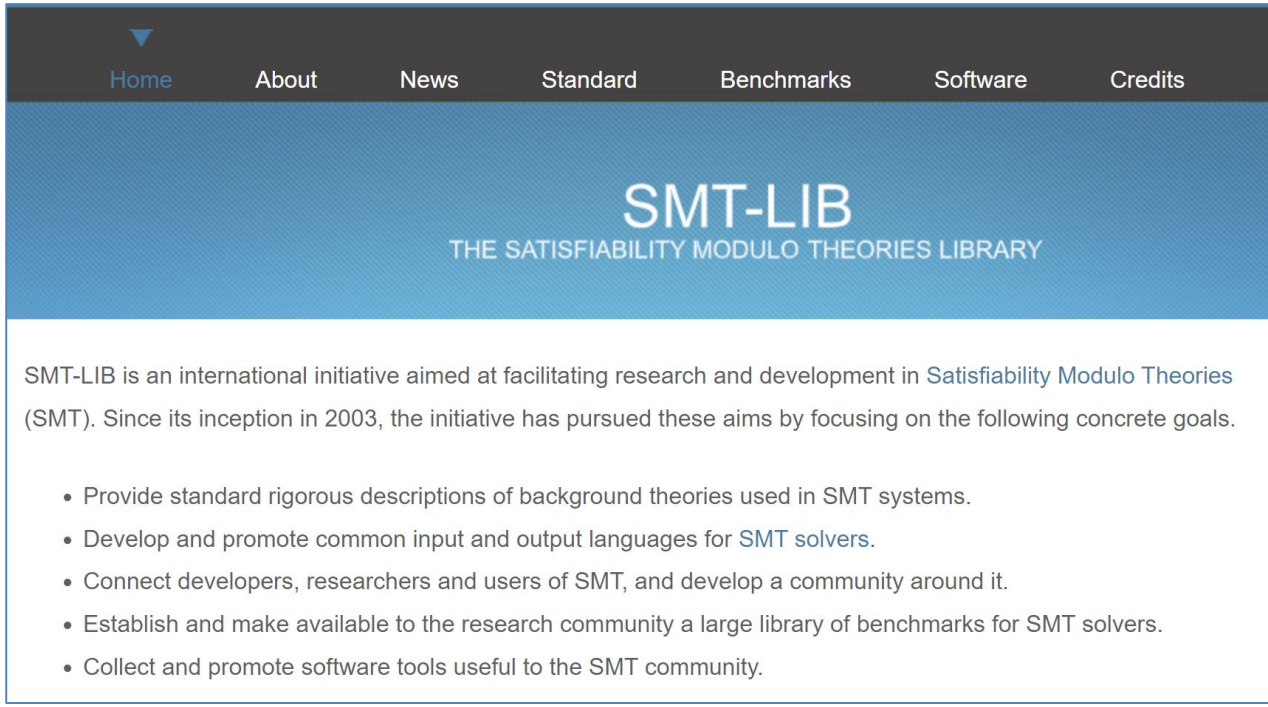
The Satisfiability Modulo Theories Library (SMT-LIB)

Moonzoo Kim

SoC. KAIST



Motivation

- How to convert the following C code into a logic formula to automatically solve by using a SMT (Satisfiability Modulo Theory) solver?

```
if( x*y + 3.14 * a[i]<<2 > x) {...}
```

1. There exist various SMT theories w/ different pros and cons

- There exists tradeoff between **expressibility** and computational **complexity**

2. Which SMT solver do you use?

a) LIA (linear integer arithmetic (i.e., not supporting bitwise operators `&`,`<<`))

b) AUFLIA (Array w/ Uninterpreted Func. and Linear Integer Arithmetic)

c) AUFLIRA (Array w/ Uninter. Func. and Linear Integer Real Arithmetic)

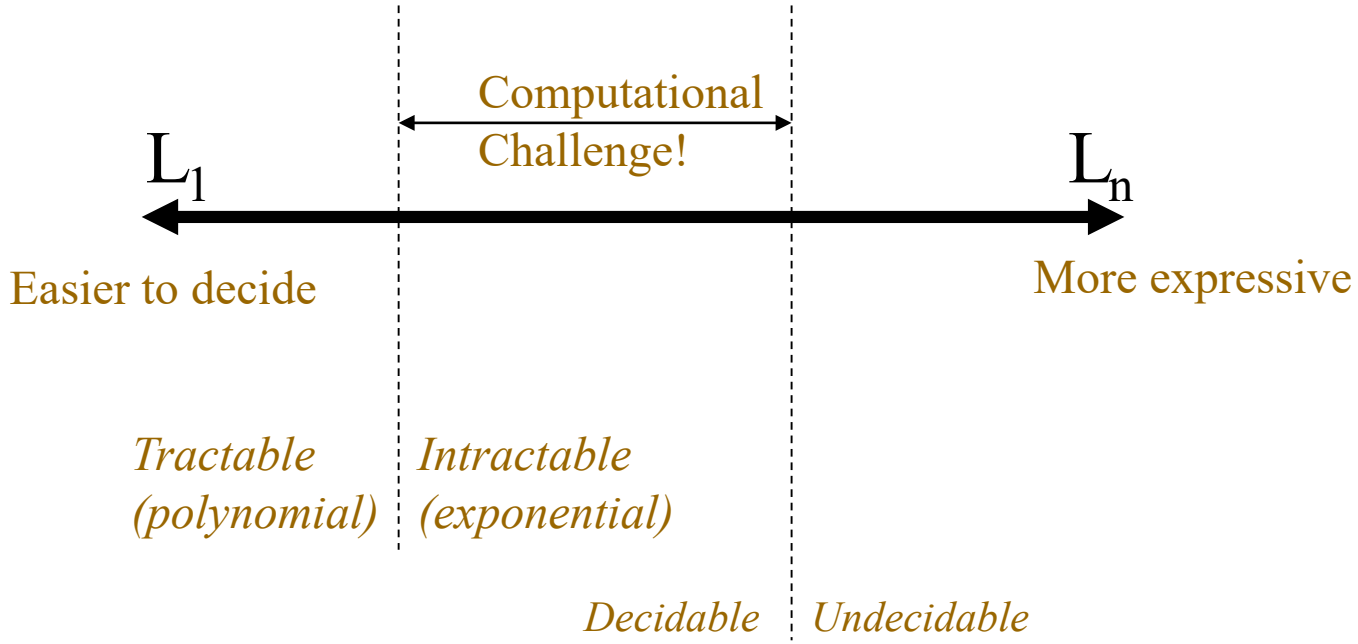
d) BV (Bit Vector)

...

2. Based on your choice of SMT solver, how to convert the C code into a logic formula?

Tradeoff: expressiveness/computational hardness.

- Assume we are given theories $L_1 \subseteq \ldots \subseteq L_n$



From the slides for the book "Decision procedures" by D.Kroening and O.Strichman

First-Order Theories

	Theory	Quantifiers Decidable	QFF Decidable
T_E	Equality	–	✓
T_{PA}	Peano Arithmetic	–	–
$T_{\mathbb{N}}$	Presburger Arithmetic	✓	✓
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	✓	✓
$T_{\mathbb{R}}$	Real Arithmetic	✓	✓
$T_{\mathbb{Q}}$	Linear Rationals	✓	✓
T_{cons}	Lists	–	✓
T^E_{cons}	Lists with Equality	–	✓

Slides from the original slides from CS156 by Prof. Z.Manna

Supported Theories (SMT-Lib v2)

- **ArraysEx**
  - Functional arrays with extensionality.
- **Fixed_Size_BitVectors**
  - Bit vectors with arbitrary size.
- **Core**
  - Core theory, defining the basic Boolean operators
- **Ints**
  - Integer numbers.
- **Reals**
  - Real numbers.
- **Reals_Ints**
  - Real and integer numbers.

Supported Sublogics

AUFLIA: Closed formulas over the theory of linear integer arithmetic and arrays extended with free sort and function symbols but restricted to arrays with integer indices and values.
AUFLIRA: Closed linear formulas with free sort and function symbols over one- and two-dimentional arrays of integer index and real value.
AUFNIRA: Closed formulas with free function and predicate symbols over a theory of arrays of arrays of integer index and real value.
LRA: Closed linear formulas in linear real arithmetic.
QF_ABV: Closed quantifier-free formulas over the theory of bitvectors and bitvector arrays.
QF_AUFBV: Closed quantifier-free formulas over the theory of bitvectors and bitvector arrays extended with free sort and function symbols.
QF_AUFLIA: Closed quantifier-free linear formulas over the theory of integer arrays extended with free sort and function symbols.
QF_AX: Closed quantifier-free formulas over the theory of arrays with extensionality.
QF_BV: Closed quantifier-free formulas over the theory of fixed-size bitvectors.
QF_IDL: Difference Logic over the integers. In essence, Boolean combinations of inequations of the form $x - y < b$ where x and y are integer variables and b is an integer constant.
QF_LIA: Unquantified linear integer arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables.
QF_LRA: Unquantified linear real arithmetic. In essence, Boolean combinations of inequations between linear polynomials over real variables.
QF_NIA: Quantifier-free integer arithmetic.
QF_NRA: Quantifier-free real arithmetic.
QF_RDL: Difference Logic over the reals. In essence, Boolean combinations of inequations of the form $x - y < b$ where x and y are real variables and b is a rational constant.
QF_UF: Unquantified formulas built over a signature of uninterpreted (i.e., free) sort and function symbols.
QF_UFBV: Unquantified formulas over bitvectors with uninterpreted sort function and symbols.
QF_UFIDL: Difference Logic over the integers (in essence) but with uninterpreted sort and function symbols.
QF_UFLIA: Unquantified linear integer arithmetic with uninterpreted sort and function symbols.
QF_UFLRA: Unquantified linear real arithmetic with uninterpreted sort and function symbols.
QF_UFNRA: Unquantified non-linear real arithmetic with uninterpreted sort and function symbols.
UFLRA: Non-linear real arithmetic with uninterpreted sort and function symbols.
UFNIA: Non-linear integer arithmetic with uninterpreted sort and function symbols.

Theory of Arrays

```
(theory Arrays
 :written_by {Silvio Ranise and Cesare Tinelli}
 :date {08/04/05}
 :sorts (Index Element Array)
 :funs ((select Array Index Element)
        (store Array Index Element Array))

 :definition
 "This is a theory of functional arrays without extensionality.
  It is formally and completely defined by the axioms below. "

 :axioms  (
  (forall (?a Array) (?i Index) (?e Element)
    (= (select (store ?a ?i ?e) ?i) ?e))

  (forall (?a Array) (?i Index) (?j Index) (?e Element)
    (or (= ?i ?j)
      (= (select (store ?a ?i ?e) ?j)  (select ?a ?j))))
)
```

Predefined data types

Predefined functions

Bounded variables

Prefix operator

```
 :notes
 "It is not difficult to prove that the two
      axioms above are logically equivalent
      to the following \"McCarthy axiom\":

 (forall  (?a Array) (?i Index) (?j Index)
          (?e Element)
        (= (select (store ?a ?i ?e) ?j)
           (ite (= ?i ?j) ?e (select ?a ?j))))

  Such an axiom appeared in the following
      paper:
  Correctness of a Compiler for Arithmetic
      Expressions,
  by John McCarthy and James Painter,
  available at http://www-formal.stanford.
      edu/jmc/mcpain.html.
 "
)
```

If-then-else term construct

Theory of Arrays w/ Extensionability

```
(theory ArraysEx
 :written_by {Silvio Ranise and Cesare Tinelli}
 :date {08/04/05}
 :updated {28/10/05}
 :history {
  Bug fix in the third axiom, pointed out by Robert
  Nieuwenhuis:
  The scope of 'forall (?i Index)' was the whole implication
  instead of just the premise of the implication.
  }
 :sorts (Index Element Array)
 :funs ((select Array Index Element)
         (store Array Index Element Array))
 :definition
 "This is a theory of functional arrays with extensionality.
  It is formally and completely defined by the axioms below.
 “
```

```
 :axioms
 (
  (forall (?a Array) (?i Index) (?e Element)
    (= (select (store ?a ?i ?e) ?i) ?e))
  (forall (?a Array) (?i Index) (?j Index) (?e Element)
    (or (= ?i ?j)
        (= (select (store ?a ?i ?e) ?j)
           (select ?a ?j))))

  (forall (?a Array) (?b Array)
     (implies (forall (?i Index) (= (select ?a ?i) (select ?b ?i)))
                 (= ?a ?b)))
 )
 :notes "This theory extends the theory Arrays with
  an axiom stating that any two arrays with the same
  elements are in fact the same array. " )
```

Theory of Integer

```
(theory Ints
:sorts (Int)
:notes
 "The (unsupported) annotations of the function/pre
      dicate symbols have
  the following meaning:
   attribute | possible value | meaning
  ------------------------------------------------------
  :assoc          //         the symbol is associative
  :comm           //         the symbol is commutative
  :unit       a constant
  :trans          //         the symbol is transitive
  :refl           //         the symbol is reflexive
  :irref          //         the symbol is irreflexive
  :antisym        //         the symbol is antisymmetric
 "
 :funs ((0 Int)
       (1 Int)
       (~ Int Int)     ; unary minus
       (- Int Int Int) ; binary minus
       (+ Int Int Int :assoc :comm :unit {0})
       (* Int Int Int :assoc :comm :unit {1})   )
 :preds ((<= Int Int :refl :trans :antisym)
        (< Int Int :trans :irref)
        (>= Int Int :refl :trans :antisym)
        (> Int Int :trans :irref)
       )
 :definition
 "This is the first-order theory of the integers, that is
, the set of all the first-order sentences over the given
signature that are true in the structure of the integer
numbers interpreting the signature's symbols in the
obvious way (with ~ denoting the negation and - the
subtraction functions). "

 :notes
 "Note that this theory is not (recursively) axiomatizable
in a first-order logic such as SMT-LIB's underlying logic.
That is why the theory is defined semantically. "
 )
```

Example of QF_LIA Benchmark

```
(set-logic QF_LIA)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const x4 Int)
(declare-const x5 Int)
;human readable form
; x1-1 >= x2 /\
; x1-3 <= x2 /\
; x1 = 2 x3+x5 /\
; x2 = 6 x4
(assert (>= (- x1 x2) 1))
(assert (<= (- x1 x2) 3))
(assert (= x1 (+ (* 2 x3) x5)))
(assert (= x2 (* 6 x4)))
(check-sat)
(get-model)
(exit)
```

Theory

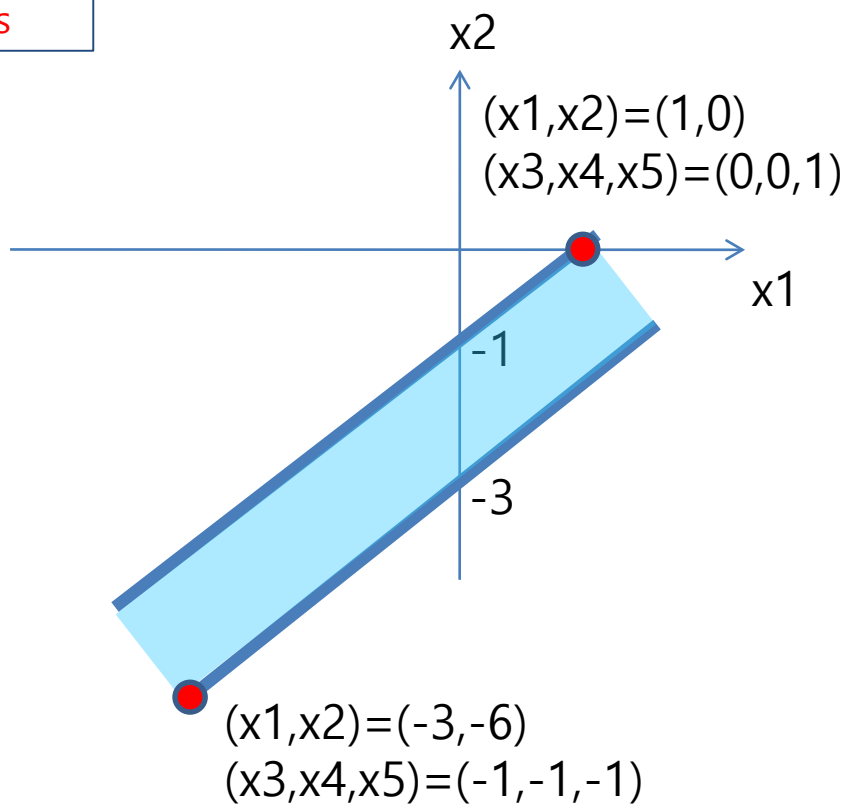
User defined variables

Comments

Target formula

Command to check satisfiability

Command to get a mode if SAT



Example of QF_UF Benchmark (1/2)

```
(set-logic QF_UF)
(declare-sort A 0)
(declare-sort B 0)

(declare-fun f (A A) B)
(declare-fun g (A B) B)
(declare-fun h1 (B A) B)
(declare-fun h2 (B B) B)

(declare-const x A)
(declare-const y B)
(declare-const w A)

;human readable form
;   g(x,y) = h1(y,x) /\
;   f(x,x) = h2(y,y) /\
;   f(x,x) != f(x,w)
(assert (= (g x y) (h1 y x)))
(assert (= (f x x) (h2 y y)))
(assert (not (= (f x x) (f x w))))
(check-sat)
(get-model)
(exit)
```

User defined data types
(0 is arity of a type)

User defined functions

If-Then-Else

```
(define-fun h1 ((x!1 B) (x!2 A)) B
  (ite (and (= x!1 B!val!0) (= x!2 A!val!0)) B!val!1
    B!val!1))

(define-fun h2 ((x!1 B) (x!2 B)) B
  (ite (and (= x!1 B!val!0) (= x!2 B!val!0)) B!val!2
    B!val!2))

(define-fun g ((x!1 A) (x!2 B)) B
  (ite (and (= x!1 A!val!0) (= x!2 B!val!0)) B!val!1
    B!val!1))

(define-fun f ((x!1 A) (x!2 A)) B
  (ite (and (= x!1 A!val!0) (= x!2 A!val!0)) B!val!2
  (ite (and (= x!1 A!val!0) (= x!2 A!val!1)) B!val!3
    B!val!2)))
```

Example of QF_UF Benchmark (2/2)

```
(set-logic QF_UF)
(declare-sort A 0)
(declare-sort B 0)

(declare-fun f (A A) B)
(declare-fun g (A B) B)
(declare-fun h1 (B A) B)
(declare-fun h2 (B B) B)

(declare-const x A)
(declare-const y B)
(declare-const w A)

;human readable form
;   g(x,y) = h1(y,x) /\
;   f(x,x) = h2(y,y) /\
;   f(x,x) != f(x,w)
(assert (= (g x y) (h1 y x)))
(assert (= (f x x) (h2 y y)))
(assert (not (= (f x x) (f x w))))
(check-sat)
(get-model)
(exit)
```

User defined data types (0 is arity of a type)

User defined functions

A model for the formula

```
x-> a0
y-> b0
w->a1
f->{(a0,a0)->b2,
      (a0,a1)->b3,
      else->b2}
g->{ (a0,b0)->b1,
       else-> b1}
h1->{(b0,a0)->b1,
       else -> b1}
h2->{(b0,b0)->b2,
       else -> b2}
```

Another Example of QF_UF Benchmark

1. Prove that

F : a=b ^ b=c -> g(f(a), b) = g(f (c), a)

is T_{QF_UF}–**satisfiable**

2. Prove

F : a=b ^ b=c -> g(f(a), b) = g(f (c), a)

is T_{QF_UF}–**valid** through proof tree

```
(set-logic QF_UF)
(declare-sort A 0)
(declare-sort B 0)
(declare-sort C 0)
(declare-const a A)
(declare-const b A)
(declare-const c A)
(declare-fun f (A) B)
(declare-fun g (B A) C)
; a=b/\ b=c ->  g(f(a),b) = g(f(c),a)
(assert (=> (and (= a b) (= b c)) (= (g ( f a) b) (g (f c) a))))
(check-sat)
(get-model)
```

Then, how to check T_{QF_UF}–validity of F by using a SMT solver???

A model for the formula

a->a0

b->a1

c->a2

f->{a0->b0, a2->b1, else->b0}

g->{(b0,a1)->c0,(b1,a0)->c1,else->c0}

Example of QF_AUFLIA

```
(set-logic QF_AUFLIA)
(declare-const data_3 (Array Int Int)) ; initial data[] declaration
; Iteration 1
(declare-const data_3_1_0 (Array Int Int))
(assert (= data_3_1_0 data_3)) ; data_(size)_(iteration)_(ssa idx)
(declare-const i_1 Int)
(assert (= i_1 0)) ; i = 0;
(declare-const j_1 Int)
(assert (= j_1 1)) ; j = 1;
(declare-const tmp_1 Int)
(assert (= tmp_1 (select data_3_1_0 0))) ; tmp = data[0];
(declare-const data_3_1_1 (Array Int Int))
(assert (= data_3_1_1 (store data_3_1_0 0 (select data_3_1_0 1)))) ;
    data[0] = data[1];
(declare-const data_3_1_2 (Array Int Int))
(assert (= data_3_1_2 (store data_3_1_1 1 tmp_1))) ; data[1] = tmp
(declare-const data_3_1_3 (Array Int Int))
; if (data[0] > data[1]) {tmp = data[0]; data[0] = data[1]; data[1] = tmp
    ;}
(assert (= data_3_1_3 (ite (> (select data_3_1_0 0) (select data_3_1_
    0 1)) data_3_1_2 data_3_1_0)))
...
(assert (not (and (<= (select data_3_3_3 0) (select data_3_3_3 1))
                  (<= (select data_3_3_3 1) (select data_3_3_3 2)))))
```

```
#define N 7
int main(){
  int data[N], i, j, tmp;
  for (i=0; i<N-1; i++)
    for (j=i+1; j<N; j++)
      if(data[i]>data[j]){
        tmp = data[i];
        data[i] = data[j];
        data[j] = tmp;
      }
  assert(data[0]<=data[1]…);
}
```

Theory of Fixed_Size_BitVectors[32]

:sorts_description
"All sort symbols of the form BitVec[i],
where i is a numeral between 1 and 32, inclusive."
:funs_description
"All function symbols with arity of the form
(concat BitVec[i] BitVec[j] BitVec[m]) where
- i,j,m are numerals
- i,j > 0
- i + j = m <= 32 "
:funs_description
"All function symbols with arity of the form
(extract[i:j] BitVec[m] BitVec[n]) where
- i,j,m,n are numerals
- 32 >= m > i >= j >= 0,
- n = i-j+1. "
:funs_description
"All function symbols with arity of the form
(op1 BitVec[m] BitVec[m]) or
(op2 BitVec[m] BitVec[m] BitVec[m]) where
- op1 is from {bvnot, bvneg}
- op2 is from {bvand,bvor,bvxor,bvsub,bvadd,bvmul}
- m is a numeral
- 0 < m <= 32 "
:preds_description
"All predicate symbols with arity of the form
(pred BitVec[m] BitVec[m]) where
- pred is from {bvult, bvule, bvuge, bvugt,
bvslt, bvsle, bvsge, bvsgt,}
- m is a numeral
- 0 < m <= 32 "

- Variables
If v is a variable of sort BitVec[m] with 0 < m <= 32, then
[[v]] is some element of [{0,...,m-1} -> {0,1}], the set of
total functions from {0,...,m-1} to {0,1}.
- Constant symbols bv0 and bv1 of sort BitVec[32]
[[bv0]] := \lambda x : [0...32). 0
[[bv1]] := \lambda x : [0...32). if x = 0 then 1 else 0
- Function symbols for concatenation
[[(concat s t)]] := \lambda x : [0...n+m).
if (x<m) then [[t]](x) else [[s]](x-m) where
s and t are terms of sort BitVec[n] and BitVec[m],
respectively, 0 < n <= 32, 0 < m <= 32, and n+m <= 32.
- Function symbols for extraction
[[(extract[i:j] s)]] := \lambda x : [0...i-j+1). [[s]](j+x)
where s is of sort BitVec[l], 0 <= j <= i < l <= 32.
- Function symbols for arithmetic operations
To define the semantics of the bitvector operators bvadd
, bvsub, bvneg, and bvmul, it is helpful to use these
ancillary functions:
o bv2nat which takes a bitvector b: [0...m) --> {0,1}
with 0 < m <= 32, and returns an integer in the range
[0...2^m), and is defined as follows:
bv2nat(b) := b(m-1)*2^{m-1} + b(m-2)*2^{m-2} + ... + b(0)*2^0
o nat2bv[m], with 0 < m <= 32, which takes a non-negative
integer n and returns the (unique) bitvector b: [0,...,m) -> {0,1}
such that b(m-1)*2^{m-1} + ... + b(0)*2^0 = n MOD 2^m
where MOD is usual modulo operation.
[[(bvadd s t)]] := nat2bv[m](bv2nat(s) + bv2nat(t))

QF_BV Example

```
; Modeling sequential code with bitvectors
; Correct swap with no temp var
; int x, y;
; x = x + y;
; y = x - y;
; x = x - y;

(set-logic QF_BV)
(set-option :produce-models true)

(declare-const x_0 (_ BitVec 32))
(declare-const x_1 (_ BitVec 32))
(declare-const x_2 (_ BitVec 32))
(declare-const y_0 (_ BitVec 32))
(declare-const y_1 (_ BitVec 32))

(assert (= x_1 (bvadd x_0 y_0)))
(assert (= y_1 (bvsub x_1 y_0)))
(assert (= x_2 (bvsub x_1 y_1)))

(assert (not  (and (= x_2 y_0) (= y_1 x_0))))
(check-sat)  ; unsat
(exit)
```