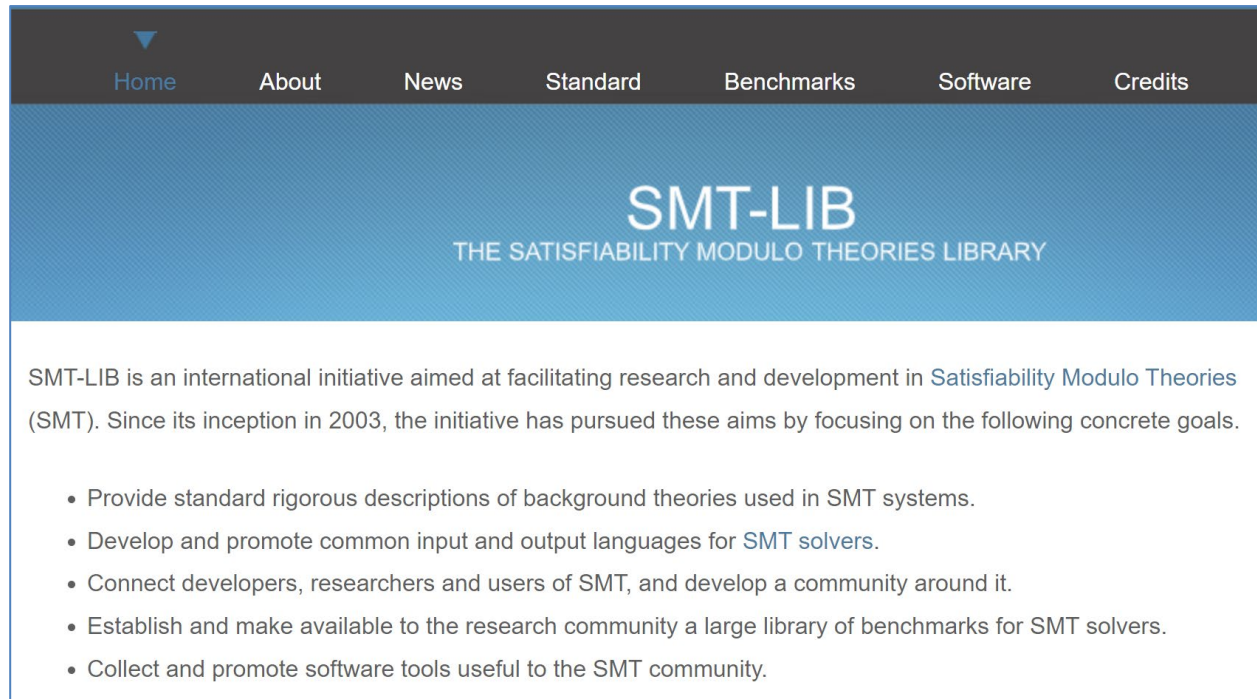


The Satisfiability Modulo Theories Library (SMT-LIB)

Moonzoo Kim
SoC. KAIST



The screenshot shows the homepage of the SMT-LIB website. At the top is a dark navigation bar with a dropdown arrow and links for Home, About, News, Standard, Benchmarks, Software, and Credits. Below this is a blue header section with the text 'SMT-LIB' and 'THE SATISFIABILITY MODULO THEORIES LIBRARY'. The main content area is white and contains a paragraph about the initiative's goals and a bulleted list of five specific goals.

▼
[Home](#) [About](#) [News](#) [Standard](#) [Benchmarks](#) [Software](#) [Credits](#)

SMT-LIB

THE SATISFIABILITY MODULO THEORIES LIBRARY

SMT-LIB is an international initiative aimed at facilitating research and development in [Satisfiability Modulo Theories \(SMT\)](#). Since its inception in 2003, the initiative has pursued these aims by focusing on the following concrete goals.

- Provide standard rigorous descriptions of background theories used in SMT systems.
- Develop and promote common input and output languages for [SMT solvers](#).
- Connect developers, researchers and users of SMT, and develop a community around it.
- Establish and make available to the research community a large library of benchmarks for SMT solvers.
- Collect and promote software tools useful to the SMT community.

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) solver
 - 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?

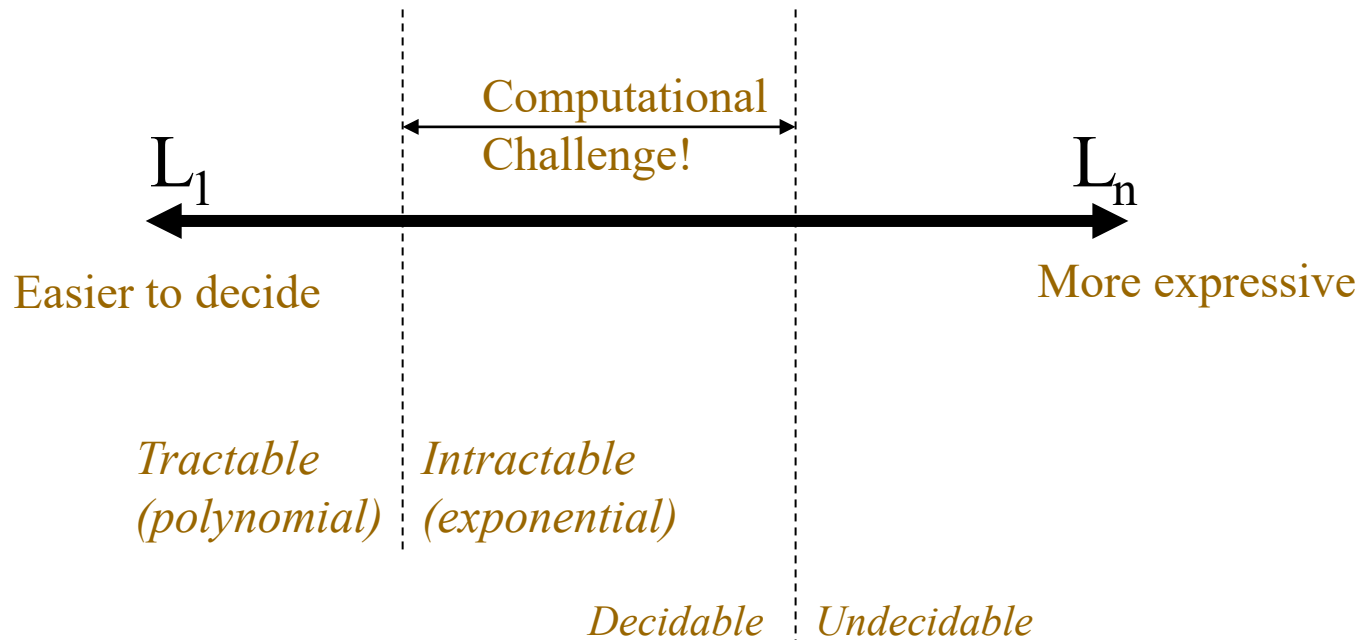
First-Order Theories

	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_{cons}^E	Lists with Equality	—	✓

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

Tradeoff: expressiveness/computational hardness.

- Assume we are given theories $L_1 \dot{\vdash} \dots \dot{\vdash} L_n$



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

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-dimensional 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)

Predefined
data types

:funs ((select Array Index Element)

(store Array Index Element Array))

Predefined f
unctions

: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))

Bounded
variables

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

(= (select (store ?a ?i ?e) ?j) (select ?a ?j))))

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))))

If-then-else
term construct

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>.

"
)

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/predicate 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)

Theory

(declare-const **x1** Int)

User defined variables

(declare-const **x2** Int)

(declare-const **x3** Int)

(declare-const **x4** Int)

(declare-const **x5** Int)

;human readable form

Comments

; $x1 - 1 \geq x2 \wedge$

; $x1 - 3 \leq x2 \wedge$

; $x1 = 2x3 + x5 \wedge$

; $x2 = 6x4$

(assert (\geq (- x1 x2) 1))

(assert (\leq (- x1 x2) 3))

(assert (= x1 (+ (* 2 x3) x5)))

(assert (= x2 (* 6 x4)))

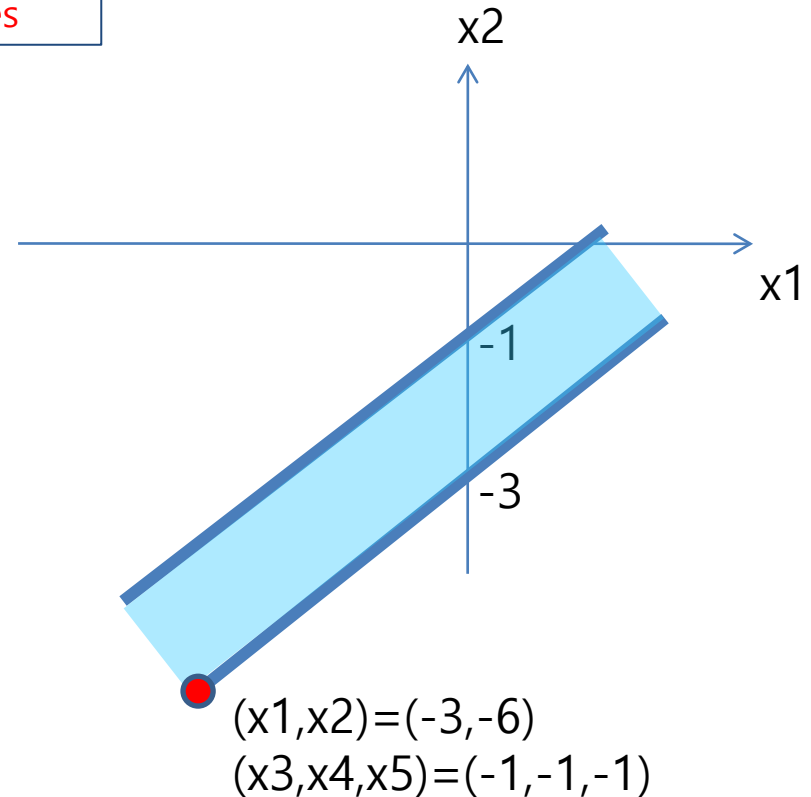
(check-sat)

Command to check satisfiability

(get-model)

Command to get a model if SAT

(exit)



Example of QF_UF Benchmark

(set-logic QF_UF)

(declare-sort **A** 0)

(declare-sort **B** 0)

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

(declare-fun **f** (**A** **A**) **B**)

(declare-fun **g** (**A** **B**) **B**)

(declare-fun **h1** (**B** **A**) **B**)

(declare-fun **h2** (**B** **B**) **B**)

User defined functions

(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)

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

$F : a=b \wedge b=c \rightarrow g(f(a), b) = g(f(c), a)$

is T_{QF_UF} —**satisfiable**

2. Prove

$F : a=b \wedge b=c \rightarrow g(f(a), b) = g(f(c), a)$

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)
```

A model for the formula

$a \rightarrow a_0$

$b \rightarrow a_1$

$c \rightarrow a_2$

$f \rightarrow \{a_0 \rightarrow b_0, a_2 \rightarrow b_1, \text{else} \rightarrow b_0\}$

$g \rightarrow \{(b_0, a_1) \rightarrow c_0, (b_1, a_0) \rightarrow c_1, \text{else} \rightarrow c_0\}$

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

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 $\text{BitVec}[i]$,
where i is a numeral between 1 and 32, inclusive."

:funs_description

"All function symbols with arity of the form
(**concat** $\text{BitVec}[i]$ $\text{BitVec}[j]$ $\text{BitVec}[m]$) where
- i, j, m are numerals
- $i, j > 0$
- $i + j = m \leq 32$ "

:funs_description

"All function symbols with arity of the form
(**extract** $[i:j]$ $\text{BitVec}[m]$ $\text{BitVec}[n]$) where
- i, j, m, n are numerals
- $32 \geq m > i \geq j \geq 0$,
- $n = i - j + 1$. "

:funs_description

"All function symbols with arity of the form
(op1 $\text{BitVec}[m]$ $\text{BitVec}[m]$) or
(op2 $\text{BitVec}[m]$ $\text{BitVec}[m]$ $\text{BitVec}[m]$) where
- op1 is from {**bvnot**, **bvneg**}
- op2 is from {**bvand**, **bvor**, **bvxor**, **bvsub**, **bvadd**, **bvmul**}
- m is a numeral
- $0 < m \leq 32$ "

:preds_description

"All predicate symbols with arity of the form
(pred $\text{BitVec}[m]$ $\text{BitVec}[m]$) where
- pred is from {**bvult**, **bvule**, **bvuge**, **bvugt**,
bvslt, **bvsle**, **bvsge**, **bvsgt**,}
- m is a numeral
- $0 < m \leq 32$ "

- Variables

If v is a variable of sort $\text{BitVec}[m]$ with $0 < m \leq 32$, then
[[v]] is some element of $\{0, \dots, m-1\} \rightarrow \{0, 1\}$, the set of
total functions from $\{0, \dots, m-1\}$ to $\{0, 1\}$.

- Constant symbols **bv0** and **bv1** of sort $\text{BitVec}[32]$

[[**bv0**]] := $\lambda x : [0 \dots 32]. 0$

[[**bv1**]] := $\lambda x : [0 \dots 32]. \text{if } x = 0 \text{ then } 1 \text{ else } 0$

- Function symbols for concatenation

[[**concat** s t]] := $\lambda x : [0 \dots n+m].$

if $(x < m)$ then [[t]](x) else [[s]]($x - m$) where

s and t are terms of sort $\text{BitVec}[n]$ and $\text{BitVec}[m]$,
respectively, $0 < n \leq 32$, $0 < m \leq 32$, and $n + m \leq 32$.

- Function symbols for extraction

[[**extract** $[i:j]$ s]] := $\lambda x : [0 \dots i - j + 1]. [[s]](j + x)$

where s is of sort $\text{BitVec}[l]$, $0 \leq j \leq i < l \leq 32$.

- 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 $b : [0 \dots m] \rightarrow \{0, 1\}$

with $0 < m \leq 32$, and returns an integer in the range

$[0 \dots 2^m]$, and is defined as follows:

$\text{bv2nat}(b) := b(m-1) \cdot 2^{m-1} + b(m-2) \cdot 2^{m-2} + \dots + b(0) \cdot 2^0$

o **nat2bv** $[m]$, with $0 < m \leq 32$, which takes a non-negative

integer n and returns the (unique) bitvector $b : [0, \dots, m] \rightarrow \{0, 1\}$

such that $b(m-1) \cdot 2^{m-1} + \dots + b(0) \cdot 2^0 = n \text{ MOD } 2^m$

where MOD is usual modulo operation.

[[**bvadd** s t]] := $\text{nat2bv}[m](\text{bv2nat}(s) + \text{bv2nat}(t))$

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)
```

Comparison between SMT-solver vs. SAT-solver

- Pros of a SMT-solver:
 - it can utilize **high-level structural information** of a target SMT formula, which will be removed in a low-level SAT formula
 - Ex. $a * (b * c) == (a * b) * c$
- Pros of a SAT-solver:
 - it has been researched for more than 60 years and achieved practical strength and reliability
 - CBMC still uses MiniSAT SAT solver as a default solver.

Performance Comparison of SMT Solvers

	Module	#L	B	#P	CVC3			Boolector			Z3		
					Size	Time	Failed	Size	Time	Failed	Size	Time	Failed
1	BubbleSort	43	35	17	9031	28.27	0	3011	1.94	0	6057	2.03	0
		43	140	17	146371	MO	1	48791	182.67	0	97722	163.15	0
2	SelectionSort	34	35	17	6982	8.48	0	1955	0.78	0	5134	0.83	0
		34	140	17	108832	MO	1	29885	74.59	0	79369	74.36	0
3	BellmanFord	49	20	33	1076	0.45	0	326	0.27	0	656	0.3	0
4	Prim	79	8	30	4008	16.88	0	1296	0.5	0	3017	0.48	0
5	StrCmp	14	1000	6	9005	9.88	0	3003	91.145	0	7006	38.75	0
6	SumArray	12	1000	7	3001	1.22	0	1001	0.93	0	2003	4.74	0
7	MinMax	19	1000	9	17989	MO	1	5997	947.58	0	11994	6.22	0
8	InsertionSort	86	35	17	9337	35.57	0	3113	2.37	0	6328	2.51	0
		86	140	17	147622	MO	1	49208	TO	1	98833	143	0
9	Fibonacci	83	15	4	16	15.12	0	16	15.6	0	16	15.2	0
10	bs	95	15	7	17	0.21	0	17	0.02	0	17	0.02	0
11	lms	258	202	23	14810	1011.92	0	5005	138.74	0	10211	138.6	0
12	Cubic	66	5	5	40	0.01	0	20	0.19	0	33	0.2	0
13	BitWise	18	8	1	77	272.38	0	27	7.51	0	53	28.37	0
14	adpcm_encode	149	41	12	6417	211.81	0	2377	738.86	0	4878	5.49	0
15	adpcm_decode	111	41	10	23885	43.77	0	9121	20.16	0	19270	14.31	0

Table 1. Results of the comparison between CVC3, Boolector and Z3. Time-outs are represented with TO in the Time column; Examples that exceed available memory are represented with MO in the Time column.

Quoted from "SMT-Based Bounded Model Checking for Embedded ANSI-C Software" by L. Cordeiro, et al ASE 2009

Performance Comparison between CBMC and SMT-CBMC

	Module	#L	B	#P	CBMC						ESW-CBMC					
					Time			#P			Time			#P		
					Encoding	Decision Procedure	Total	Passed	Violated	Failed	Encoding	Decision Procedure	Total	Passed	Violated	Failed
1	sensor	603	5	167	2.04	0.002	2.04	167	0	0	1.23	0.02	1.26	167	0	0
2	crc	125	257	18	5.60	0.003	5.60	18	0	0	4.08	0.07	4.16	18	0	0
3	fft1	218	9	72	0.44	0.001	0.44	72	0	0	0.43	0.005	0.43	72	0	0
4	fft1k	155	1025	39	MO	MO	MO	0	0	39	2337.83	0.055	2337.88	39	0	0
5	fibcall	83	30	2	0.19	0	0.19	2	0	0	0.15	0.002	0.15	2	0	0
6	fir	314	34	25	4.88	0.02	4.9	25	0	0	3.36	0.68	4.04	25	0	0
7	insertsort	86	10	17	0.36	0.005	0.37	17	0	0	0.31	0.02	0.32	17	0	0
8	jfdctint	374	65	331	1.22	0.001	1.22	330	1	0	0.45	2.41	2.86	330	1	0
9	lms	258	202	35	MO	MO	MO	0	0	35	132.6	0.24	132.84	35	0	0
10	ludcmp	144	7	88	4.52	TO	TO	87	0	1	0.017	1.44	1.46	88	0	0
11	matmul	81	6	31	1.16	0	1.16	31	0	0	1.06	0.012	1.07	31	0	0
12	qurt	164	20	8	18.83	TO	TO	7	0	1	1.22	7.7	8.92	8	0	0
13	bcnt	86	17	162	4.42	0.05	4.47	162	0	0	1.24	0.89	2.13	162	0	0
14	blit	95	1	129	0.21	0.001	0.21	128	1	0	0.13	0.28	0.41	128	1	0
15	pocsag	521	42	183	15.32	0.1	15.42	182	1	0	12.33	5.77	18.1	182	1	0
16	adpcm	473	100	553	74.34	3.52	77.86	553	0	0	45.73	9.24	54.97	553	0	0
17	laplace	110	11	76	30.81	TO	TO	0	0	76	12.32	0.29	12.62	76	0	0
18	exStbKey	558	20	18	1.23	0.002	1.23	18	0	0	1.22	0.004	1.23	18	0	0
19	exStbHDMI	1045	15	25	167.91	78.97	246.88	25	0	0	164.43	33.53	197.96	25	0	0
20	exStbLED	430	40	6	195.97	129.8	325.77	6	0	0	165.63	44.53	210.16	6	0	0
21	exStbHwAcc	1432	1000	113	0.67	0.002	0.67	113	0	0	0.72	0.004	0.73	113	0	0
22	exStbResolution	353	200	40	271.8	319.13	590.93	40	0	0	269.31	1161.16	1430.47	40	0	0