SMT-LIB 3: Bringing higher-order logic to SMT

Clark Barrett Pascal Fontaine Cesare Tinelli

Stanford University, USA

The University of Iowa, USA

Université de Lorraine, CNRS, Inria, LORIA, France

Disclaimer

Many things here

- are early work in progress
- are inconsistent with each other
- need to be concretely applied to reveal flaws
- have not been properly discussed with the SMT community
- will evolve

Credits

Based on inputs from
Nikolaj Bjørner,
Jasmin Blanchette,
Koen Claessen,
Tobias Nipkow,
. . . ,
[your name here!]

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers
- around 300,000 benchmarks

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers
- around 300,000 benchmarks
- Command language
  - Stack-based, tell-and-ask execution model
  - Benchmarks are command scripts
  - Same online and offline behavior

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers
- around 300,000 benchmarks
- Command language
  - Stack-based, tell-and-ask execution model
  - Benchmarks are command scripts
  - Same online and offline behavior
- Simple syntax
  - Sublanguage of Common Lisp S-expressions
  - Easy to parse
  - Few syntactic categories

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers
- around 300,000 benchmarks
- Command language
  - Stack-based, tell-and-ask execution model
  - Benchmarks are command scripts
  - Same online and offline behavior
- Simple syntax
  - Sublanguage of Common Lisp S-expressions
  - Easy to parse
  - Few syntactic categories
- Powerful underlying logic
  - Many sorted FOL with (pseudo-)parametric types
  - Schematic theory declarations
  - Semantic definition of theories

SMT-LIB 2 Concrete Syntax

Strict subset of Common Lisp S-expressions:

$$
\begin{array}{lcl}
\langle \mathit{spec\_constant} \rangle & ::= & \langle \mathit{numeral} \rangle \mid \langle \mathit{decimal} \rangle \\
& \mid & \langle \mathit{hexadecimal} \rangle \mid \langle \mathit{binary} \rangle \\
& \mid & \langle \mathit{string} \rangle \\
\\
\langle \mathit{s\_expr} \rangle & ::= & \langle \mathit{spec\_constant} \rangle \mid \langle \mathit{symbol} \rangle \\
& \mid & (\ \langle \mathit{s\_expr} \rangle^{*}\ )
\end{array}
$$

Example: Concrete Syntax

```
(declare-datatype List (par (X) (
  (nil)
  (cons (head X) (tail (List X))) )))

(declare-fun append ((List Int) (List Int) (List Int)))

(declare-const a Int)

(assert
  (forall ((x (List Int)) (y (List Int)))
    (= (append x y)
       (ite (= x (as nil (List Int)))
         y
         (let ((h (head x)) (t (tail x)))
           (cons h (append t y)) )))))

(assert (= (append (cons a (as nil (List Int)))
           (append (cons 2 (as nil (List Int))) nil)))

(check-sat)
```

Example: TIP vs. SMT-LIB

```
(declare-datatypes () ((Nat (Zero) (Succ (pred Nat)))))
(define-fun-rec
  plus
    ((x Nat) (y Nat)) Nat
    (match x
      (case Zero y)
      (case (Succ n) (Succ (plus n y)))))
(assert-not (forall ((n Nat) (m Nat))
              (= (plus n m) (plus m n))))
(check-sat)
```

Example: TIP vs. SMT-LIB

```
(declare-datatype Nat ((Zero) (Succ (Pred Nat))))
(define-fun-rec
  plus
    ((x Nat) (y Nat)) Nat
    (match x
      (Zero y)
      ((Succ n) (Succ (plus n y)))))
(assert (not (forall ((n Nat) (m Nat))
               (= (plus n m) (plus m n)))))
(check-sat)
```

SMT-LIB vs. TIP

Many TIP features have been integrated into the SMT-LIB

- `declare-datatypes`: similar semantics, simplified syntax
- `match` (Section 3.5.1)
  - No `case` keyword
  - No `default` keyword: use variable (usable inside term)
- `define-fun-rec` was already in SMT-LIB 2.5 (Section 4.2.3)
- `assert-not`: Tagging a goal can be done with `:named` annotation
- `par`: parametric functions are not yet supported

TIP

This document does not yet cover mutual recursion (over data types or over functions), or partial branches and partiality.

- SMT-LIB does cover mutual recursion, over functions and data-types: `define-funs-rec` and `declare-datatypes`
- partiality is not covered

From Many-sorted FOL to HOL

Motivation:

- Several hammers for ITP systems use SMT solvers
- New communities are extending SMT-LIB with HOL features (for synthesis, inductive reasoning, symbolic computation, ...)

Goals:

- Serve these new users and other non-traditional users
- Maintain backward compatibility as much as possible

From Many-sorted FOL to HOL

Plan:

- Adopt (Gordon's) HOL with parametric types, rank-1 polymorphism, and extensional equality
- Extend syntax by introducing $\rightarrow$ type, λ and ε binders
- Make all function symbols Curried
- Enable higher-order quantification
- Keep SMT-LIB 2 constructs/notions but define them in terms of HOL

SMT-LIB 3: Basic Concrete Syntax for Sorts (Types)

$$
\begin{array}{llcl}
 & \langle \textit{indentifier} \rangle & ::= & \langle \textit{symbol} \rangle \mid \texttt{(\_} \ \langle \textit{symbol} \rangle \ \langle \textit{label} \rangle^{+} \ \texttt{)} \\
\sigma & \langle \textit{sort} \rangle & ::= & \langle \textit{identifier} \rangle \\
 & & \mid & \texttt{(->} \ \langle \textit{sort} \rangle^{+} \ \langle \textit{sort} \rangle \ \texttt{)} \\
 & & \mid & \texttt{(} \ \langle \textit{identifier} \rangle \ \langle \textit{sort} \rangle^{+} \ \texttt{)} \\
\tau & \langle \textit{par\_sort} \rangle & ::= & \langle \textit{sort} \rangle \\
 & & \mid & \texttt{(par (} \ \langle \textit{symbol} \rangle^{+} \ \texttt{)} \ \langle \textit{sort} \rangle \ \texttt{)}
\end{array}
$$

`->` predefined right-associative type constructor

SMT-LIB 3: Basic Concrete Syntax for Terms

```
⟨sorted_var⟩   ::=   ( ⟨symbol⟩ ⟨sort⟩ )

⟨term⟩         ::=   ⟨spec_constant⟩
                 |   ⟨identifier⟩
                 |   ( ⟨term⟩ ⟨term⟩ )
                 |   ( lambda ( ⟨sorted_var⟩ ) ⟨term⟩ )
                 |   ( choose ( ⟨sorted_var⟩ ) ⟨term⟩ )
                 |   ( ! ⟨term⟩ ⟨attribute⟩+ )
```

SMT-LIB 3: Extended Concrete Syntax for Terms

- $(t_1 \;\; t_2 \;\; t_3 \;\; \cdots \;\; t_n) := ((t_1 \;\; t_2) \;\; t_3 \;\; \cdots \;\; t_n)$

SMT-LIB 3: Extended Concrete Syntax for Terms

- $(t_1\ \ t_2\ \ t_3\ \ \cdots\ \ t_n) := ((t_1\ \ t_2)\ \ t_3\ \ \cdots\ \ t_n)$
- $(\texttt{lambda}\ ((x\ \ \sigma)\ \ (x_1\ \ \sigma_1)\ \ \cdots\ \ (x_n\ \ \sigma_n))\ \ \varphi) :=$

  $(\texttt{lambda}\ ((x\ \ \sigma))$

  $\quad(\texttt{lambda}\ ((y_1\ \ \sigma_1)\ \ \cdots\ \ (y_n\ \ \sigma_n))\ \ \varphi[y_i/x_i]))$ with y_i fresh

SMT-LIB 3: Extended Concrete Syntax for Terms

- `(`t_1 t_2 t_3 $\cdots$ t_n`)` := `((`t_1 t_2`)` t_3 $\cdots$ t_n`)`
- `(lambda ((`x σ`) (`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`))` φ`)` :=
  `(lambda ((`x σ`))`
  `(lambda ((`y_1 σ_1`)` $\cdots$ `(`y_n σ_n`))` $\varphi[y_i/x_i]$`))` with y_i fresh
- `(let ((`x_1 t_1`)` $\cdots$ `(`x_n t_n`))` t`)` :=
  `((lambda ((`x_1 σ_1`)` $\cdots$ `(`x_n σ_n`))` t`)` t_1 $\cdots$ t_n`)`
  where σ_i is the sort of t_i

SMT-LIB 3: Extended Concrete Syntax for Terms

- $(t_1\ t_2\ t_3\ \cdots\ t_n) := ((t_1\ t_2)\ t_3\ \cdots\ t_n)$

- $(\texttt{lambda}\ ((x\ \sigma)\ (x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$

  $(\texttt{lambda}\ ((x\ \sigma))$

  $(\texttt{lambda}\ ((y_1\ \sigma_1)\ \cdots\ (y_n\ \sigma_n))\ \varphi[y_i/x_i]))$ with y_i fresh

- $(\texttt{let}\ ((x_1\ t_1)\ \cdots\ (x_n\ t_n))\ t) :=$

  $((\texttt{lambda}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ t)\ t_1\ \cdots\ t_n)$

  where σ_i is the sort of t_i

- $(\texttt{forall}\ ((x\ \sigma))\ \varphi) :=$

  $(=\ (\texttt{lambda}\ ((x\ \sigma))\ \varphi)\ (\texttt{lambda}\ ((x\ \sigma))\ \texttt{true}))$

  $(\texttt{forall}\ ((x_1\ \sigma_1)\ (x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$

  $(\texttt{forall}\ ((x_1\ \sigma_1))\ (\texttt{forall}\ ((x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi))$

SMT-LIB 3: Extended Concrete Syntax for Terms

- $(t_1\ t_2\ t_3\ \cdots\ t_n) := ((t_1\ t_2)\ t_3\ \cdots\ t_n)$
- $(\texttt{lambda}\ ((x\ \sigma)\ (x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$
  $(\texttt{lambda}\ ((x\ \sigma))$
  $(\texttt{lambda}\ ((y_1\ \sigma_1)\ \cdots\ (y_n\ \sigma_n))\ \varphi[y_i/x_i]))$ with y_i fresh
- $(\texttt{let}\ ((x_1\ t_1)\ \cdots\ (x_n\ t_n))\ t) :=$
  $((\texttt{lambda}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ t)\ t_1\ \cdots\ t_n)$
  where σ_i is the sort of t_i
- $(\texttt{forall}\ ((x\ \sigma))\ \varphi) :=$
  $(=\ (\texttt{lambda}\ ((x\ \sigma))\ \varphi)\ (\texttt{lambda}\ ((x\ \sigma))\ \texttt{true}))$

  $(\texttt{forall}\ ((x_1\ \sigma_1)\ (x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$
  $(\texttt{forall}\ ((x_1\ \sigma_1))\ (\texttt{forall}\ ((x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi))$
- $(\texttt{choose}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) := \ldots$

SMT-LIB 3: Extended Concrete Syntax for Terms

- $(t_1\ t_2\ t_3\ \cdots\ t_n) := ((t_1\ t_2)\ t_3\ \cdots\ t_n)$
- $(\texttt{lambda}\ ((x\ \sigma)\ (x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$
  $(\texttt{lambda}\ ((x\ \sigma))$
  $(\texttt{lambda}\ ((y_1\ \sigma_1)\ \cdots\ (y_n\ \sigma_n))\ \varphi[y_i/x_i]))$ with y_i fresh
- $(\texttt{let}\ ((x_1\ t_1)\ \cdots\ (x_n\ t_n))\ t) :=$
  $((\texttt{lambda}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ t)\ t_1\ \cdots\ t_n)$
  where σ_i is the sort of t_i
- $(\texttt{forall}\ ((x\ \sigma))\ \varphi) :=$
  $(\texttt{=}\ (\texttt{lambda}\ ((x\ \sigma))\ \varphi)\ (\texttt{lambda}\ ((x\ \sigma))\ \texttt{true}))$

  $(\texttt{forall}\ ((x_1\ \sigma_1)\ (x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi) :=$
  $(\texttt{forall}\ ((x_1\ \sigma_1))\ (\texttt{forall}\ ((x_2\ \sigma_2)\ \cdots\ (x_n\ \sigma_n))\ \varphi))$
- $(\texttt{choose}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) := \ldots$
- $(\texttt{exists}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi) := \ldots$

SMT-LIB 3: Commands

- As in SMT-LIB 2
- Fed to the solver's standard input channel or stored in a file
- Look like Lisp function calls: (⟨*comm_name*⟩ ⟨*arg*⟩*)
- Operate on an stack of *assertion sets*
- Cause solver to outputs an S-expression to the standard output/error channel
- Four categories:
  - *assertion-set* commands, modify the assertion set stack
  - *post-check* commands, query about the assertion sets
  - *option* commands, set solver parameters
  - *diagnostic* commands, get solver diagnostics

SMT-LIB 3: Assertion-Set Commands

`(declare-sort` s n`)`

Example: `(declare-sort Elem 0)`
`(declare-sort Set 1)`

Effect: declares sort symbol s with arity n

`(define-sort` s `(`$u_1 \cdots u_n$`)` σ`)`

Example: `(define-sort MyArray (u) (Array Int u))`

Effect: enables the use of `(MyArray Real)`
as a *shorthand* for `(Array Int Real)`

SMT-LIB 3: Assertion-Set Commands

`(declare-const` f τ`)`

Example:

```
(declare-const a (Array Int Real))
(declare-const g (-> Int Int Int))
(declare-const len (par (X) (-> (List X) Int)))
```

Effect: declares f with type τ

`(declare-fun` f `(`$\sigma_1 \cdots \sigma_n$`)` σ`)`

Example:

```
(declare-fun a () (Array Int Real))
(declare-fun g (Int Int) Int)
```

Effect: same as `(declare-const` f `(->` $\sigma_1 \cdots \sigma_n$ σ`))`

`(declare-fun` f `(par (`$u_1 \cdots u_n$`)` `(`$\sigma_1 \cdots \sigma_n$`)` σ`))`

Example:

```
(declare-fun len (par (X) ((List X)) Int))
```

Effect: same as
`(declare-const` f `(par (`$u_1 \cdots u_n$`)` `(->` $\sigma_1 \cdots \sigma_n$ σ`)))`

SMT-LIB 3: Assertion-Set Commands

`(set-logic` s`)` deprecated!

SMT-LIB 3: `set-logic` replacements

`(import-sorts` T $[($`$\sigma_1 \cdots \sigma_n$`$)]$`)`

Example:
```
(import-sort Arrays)
(import-sort Reals_Int (Real Int))
(import-sort Arrays ((par (X) (Array Int X))))
```

Effect: Import all instances of sorts $[\sigma_1 \cdots \sigma_n]$ in theory T

`(deport-sorts` T `(`$\sigma_1 \cdots \sigma_n$`))`

Example:
```
(deport-sort Reals_Int (Real))
(deport-sort Arrays
  ((par (X Y) (Array Int (Array X Y)))))
```

Effect: Remove for imported sort set all instances of sorts $\sigma_1 \cdots \sigma_n$ in theory T

SMT-LIB 3: `set-logic` replacements

`(import-funs` T $[($f_1 \cdots f_n$$)]$`)`

Example:
```
(import-funs Arrays)
(import-funs Reals_Int (- NUMERALS (+ Int Int Int)))
(import-funs Arrays
  ((par (X) (store Int (Array Int X) X))))
```

Effect: Import all instances of function symbols $f_1 \cdots f_n$ in theory T over imported sorts

`(deport-funs` T `(`$f_1 \cdots f_n$`))`

Example:
```
(deport-fun Reals_Int (/ div mod *))
(deport-fun Arrays (store))
```

Effect: disable all instances of function symbols $f_1 \cdots f_n$ in theory T over imported sorts

SMT-LIB 3: `set-logic` replacements

`(enable (`$l_1 \cdots l_n$`))`

Example: `(enable (order-1 user-declarations datatypes))`
`(enable (order-1 closures quantifiers))`
`(enable (order-2 quantifiers))`

Effect: enable the listed syntactic features

`(disable (`$l_1 \cdots l_n$`))`

Example: `(disable (closures choice))`
`(disable (recursive-definitions quantifiers)))`

Effect: disable the listed syntactic features

Conclusion

- Most TIP features are or will be included in SMT-LIB
- A more modular presentation of the format (extensions)
- Better handling of combination of theories
- What next?