SMT-LIB 3

Clark Barrett   Pascal Fontaine   Cesare Tinelli

Stanford University, USA

Université de Lorraine, CNRS, Inria, LORIA, France

The University of Iowa, USA

Disclaimer

This

- is work in progress
- needs discussion
- needs to be concretely applied to reveal flaws
- needs to be discussed with the SMT community
- will most likely evolve

We have been discussing SMT-LIB 3 for several years.
It is getting clearer in our mind. It will eventually come.

Two main goals:

- from FOL to HOL
- find a nice replacement for logics

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 340,000 benchmarks

SMT-LIB 2 Standard

- Widely adopted I/O language for SMT solvers
- Around 340,000 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 340,000 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 340,000 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 (= (cons a (as nil (List Int)))
           (append (cons 2 (as nil (List Int))) nil)))

(check-sat)
```

From Many-sorted FOL to HOL

Motivation:

- Several hammers for ITP systems use SMT solvers
- 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)

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

- `->` predefined right-associative type constructor
- `as` will probably have a new, simpler semantics

SMT-LIB 3: Basic Concrete Syntax for Terms

$$
\begin{array}{lcl}
\langle \textit{sorted\_var} \rangle & ::= & \texttt{(} \ \langle \textit{symbol} \rangle \ \langle \textit{sort} \rangle \ \texttt{)} \\[1em]
\langle \textit{term} \rangle & ::= & \langle \textit{spec\_constant} \rangle \\
& | & \langle \textit{identifier} \rangle \\
& | & \texttt{(} \ \langle \textit{term} \rangle \ \langle \textit{term} \rangle \ \texttt{)} \\
& | & \texttt{(} \ \texttt{lambda} \ \texttt{(} \ \langle \textit{sorted\_var} \rangle \ \texttt{)} \ \langle \textit{term} \rangle \ \texttt{)} \\
& | & \texttt{(} \ \texttt{choose} \ \texttt{(} \ \langle \textit{sorted\_var} \rangle \ \texttt{)} \ \langle \textit{term} \rangle \ \texttt{)} \\
& | & \texttt{(} \ \texttt{!} \ \langle \textit{term} \rangle \ \langle \textit{attribute} \rangle^{+} \ \texttt{)} \\[1em]
\langle \textit{par\_term} \rangle & ::= & \texttt{(} \ \texttt{par} \ \texttt{(} \ \langle \textit{symbol} \rangle^{+} \ \texttt{)} \ \langle \textit{term} \rangle \ \texttt{)} \\
& | & \texttt{(} \ \texttt{not} \ \langle \textit{par\_term} \rangle \ \texttt{)} \\
& | & \langle \textit{term} \rangle
\end{array}
$$

- `par` is used in terms too. Should this be renamed `forall-type`?

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

- $(t_1\ t_2\ t_3\ \cdots\ t_n)$:= $((t_1\ t_2)\ t_3\ \cdots\ t_n)$
- $(\texttt{lambda}\ ((x\ \sigma)\ (x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ \varphi)$:=
  $(\texttt{lambda}\ ((x\ \sigma))$
  $(\texttt{lambda}\ ((y_1\ \sigma_1)\ \cdots\ (y_n\ \sigma_n))\ \varphi[y_i/x_i]))$ 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

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
- $(\texttt{let}\ ((x_1\ t_1)\ \cdots\ (x_n\ t_n))\ t) :=$
  $((\texttt{lambda}\ ((x_1\ \sigma_1)\ \cdots\ (x_n\ \sigma_n))\ t)\ t_1\ \cdots\ t_n)$
  where σ_i is the sort of t_i
- $(\texttt{forall}\ ((x\ \sigma))\ \varphi) :=$
  $(\texttt{=}\ (\texttt{lambda}\ ((x\ \sigma))\ \varphi)\ (\texttt{lambda}\ ((x\ \sigma))\ \texttt{true}))$

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

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

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

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

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

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

- Mostly as in SMT-LIB 2
- Fed to the solver's standard input channel or stored in a file
- Look like Lisp function calls: ($\langle$*comm_name*$\rangle$ $\langle$*arg*$\rangle^*$)
- 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

`(assert` t`)` where $t \in \langle \textit{par\_term} \rangle$

`(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`)`

- We want logic to be parsable.
- Rationale: there are too many of them, and need for more.
- Need to better express combinations of theories.

SMT-LIB 3: Theories

- Theories as written in SMT-LIB 2.6 are a reminiscence of SMT-LIB 1
- They are never parsed (and only partially parsable)
- What should a theory be? A script? Something totally informal?
- We are probably going for something in-between
- Use SMT-LIB 3 commands when possible, informal descriptions otherwise

SMT-LIB 3: Logics (1/2)

- New, more general concept of module
- Modules are parsable scripts, possibly including classical SMT commands
- It is not mandatory that your solver parse them: you might have hard-coded modules inside your solver
- Enable certain features (to be defined), e.g.
  `(enable (order-1 datatypes))`
- Import theories: `import` command imports a theory, possibly with attributes specifying restrictions defined in theory itself
  `(import Integers :linear)`

SMT-LIB 3: Logics (2/2)

- Handle namespaces: imported theories add symbols within a namespace
- `open` moves namespaced symbols to the global space
  `(open Sets)`
- Allow dynamic overloading of symbols while importing
- The idea would be that modules define current logics as precisely as possible (and could be used for other things).

Modules and their details are still w.i.p.

Conclusion

- Fully backward compatible
- Extension to HOL
- Better handling of combination of theories
- Cleaning “logics”
- What next?