The SMT-LIB Standard: Version 1.2

Silvio Ranise

LORIA and
INRIA-Lorraine
Nancy, France
`ranise@loria.fr`

Cesare Tinelli

Computer Science Department
The University of Iowa
Iowa City, IA, USA
`tinelli@cs.uiowa.edu`

Release date: **5 Aug 2006**

Abstract

The SMT-LIB initiative is an international effort, coordinated by these authors and supported by several research groups world-wide, with the main goal of producing an extensive on-line library of benchmarks for *satisfiability modulo theories*. This paper defines syntax and semantics of the language used by SMT-LIB for writing theory specifications and benchmarks.

Contents

List of Figures

0   About this Document

This paper defines and discusses Version 1.2 of a proposed language standard for the Satisfiability Modulo Theories Library, or SMT-LIB for short.

This is an incomplete working document that is updated on a semi-regular basis. Each new release of the document is identified by its date. Each new release of the same version of the SMT-LIB format will contain only *conservative* additions and changes with respect to the previous release, that is, it will not modify any already given definitions for the SMT-LIB format. A new release only adds missing definitions, fixes errors, or provides a more complete and improved presentation of the format.

New versions of the SMT-LIB format, on the other hand, may contain non-conservative changes with respect to the previous version.

To facilitate the reading of the document, explanations of the rationale of the various design decisions taken in defining the SMT-LIB format are written in small sans-serif font. They can be skipped on a first reading.

Since this is a working draft, it may have missing parts. Place holders for missing material are denoted by text in square brackets.

0.1   Copyright Notice

1   Introduction

The main goal of the SMT-LIB initiative [2], coordinated by these authors, is to produce a on-line library of benchmarks for *Satisfiability Modulo Theories*.

By benchmark we mean a logical formula to be checked for satisfiability with respect to (combinations of) background theories of interest. Examples of background theories typically used in computer science are the theory of real numbers, the theory of integer numbers, and the theories of various data structures such as lists, arrays, bit vectors and so on.

A lot of work has been done in the last few years by several research groups on building systems for satisfiability modulo theories [**?**]. The main motivation of the SMT-LIB initiative is the expectation that having a library of benchmarks will greatly facilitate the evaluation and the comparison of these systems, and advance

the state of the art in the field, in the same way as, for instance, the TPTP library [3] has done for theorem proving, or the SATLIB library [1] has done for propositional satisfiability.

The SMT-LIB initiative is endorsed by a large number of research groups worldwide, and is financially supported in part by industry and governmental institutions (see Section 8).

SMT-LIB consists conceptually of two main sections: one containing the specification of several background theories and logics (see later), and another containing benchmarks, grouped under a number of indexes such as their corresponding background theory/logic, the class of formulas they belong to, the type of problem they originate from and so on.

For the library to be viable and useful, SMT-LIB adopts a common format for expressing the benchmarks, and for defining the background theories in a rigorous way—so that there is no doubt on which theories are intended. Future work on the library might also include a format for specifying unsatisfiability proofs for benchmarks and one for describing models.

2 Basic Assumptions and Structure

In the SMT-LIB format, input problems, i.e. logical formulas, are assumed to be checked for satisfiability, not validity.[1] In particular, given a theory T and a formula φ, the problem of interest is whether φ is *satisfiable in T*, or is *satisfiable modulo T*, that is, whether there is a model of T that satisfies (the existential closure of) φ.

Informally speaking, SMT-LIB calls an *SMT solver*, or also a *satisfiability procedure*, any procedure for satisfiability modulo some given theory. In general, with satisfiability procedures one can distinguish among

1. a procedure's *underlying logic* (e.g., first-order, modal, temporal, second-order, and so on),

2. a procedure's *background theory*, the theory against which satisfiability is checked, and

3. a procedure's *input language*, the class of formulas the procedure accepts as input.

For instance, in a solver for linear arithmetic the underlying logic is first-order logic with equality, the background theory is the theory of real numbers, and the

---

[1] Note that the difference matters only for those classes of problems that are not closed under logical negation.

input language is often limited to conjunctions of inequations between linear polynomials.

For better clarity and modularity, the three aspects above are kept separate in SMT-LIB. SMT-LIB's commitments to each of them are described in the following.

2.1 Underlying Logic

Version 1.2 of the SMT-LIB format adopts as its underlying logic a basic many-sorted version of first-order logic with equality. This logic allows the definition of sorts and of sorted symbols but does not allow more sophisticated constructs such as subsorts, sort constructors, explicit sort declarations for terms, and so on. There is a plan to add some of these features will be added in future versions but only as needed, in response to extension requests from the SMT community[2].

In an attempt to combine the simplicity and familiarity of classical (i.e., unsorted) first-order logic with the convenience of a sorted language, the SMT-LIB format currently defines two semantics for its underlying logic: the first one is a translation semantics into classical first order logic with equality ($FOL^=$), the second one is a direct algebraic semantics based on many-sorted models.

The first semantics is meant to ease the transition to a sorted framework for existing tools, expertise and results, most of which have so far been developed in the context of classical $FOL^=$. For the purposes of SMT-LIB the two semantics are equivalent, as an SMT-LIB formula admits a (many-sorted) model in the algebraic semantics if and only if its translation into $FOL^=$ admits a (unsorted) model in $FOL^=$.

The logic and its two semantics are specified in the rest of this document.

2.2 Background Theories

One of the goals of the SMT-LIB initiative is to clearly define a catalog of background theories, starting with a small number of popular ones, and adding new ones as solvers for them are developed. Theories are specified in SMT-LIB independently of any benchmarks or solvers. Each benchmark then contains a reference to its own background theory.

The SMT-LIB format will eventually distinguish between *basic* (or *component*) theories and *combined* (or *structured*) theories. Basic theories will include theories such as the theory of real numbers, the theory of arrays, the theory of lists and so on. A combined theory will be one that is defined as a combination of basic theories.

---

[2] For instance, it is expected that the next version of the format will include HOL-style [?] parametric types.

The current version of SMT-LIB supports only basic theories. This means in practice that a theory composed of previously defined ones can always be defined in this version of the format, but only from scratch, as if it were a basic theory, with no external references to its previously defined components. Support for the modular definition of structured theories will be added in future versions of the format.

For practicality, the format insists that only the signature of a theory be specified formally.[3] The theory itself can be defined either formally, by means of a set of axioms, or informally, in natural language, as convenient.

2.3 Background Logics

The SMT-LIB format adopts a single and general first-order (sorted) language in which to write all SMT-LIB benchmarks.

It is often the case, however, that many benchmarks are expressed in some fragment of the language of first-order logic. The particular fragment in question does matter because one can often write a solver specialized on that fragment that is a lot more efficient than a solver meant for a larger fragment.[4]

An extreme case of this situation occurs when satisfiability modulo a given theory T is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays [?]. But a similar situation occurs even when the decidability of the satisfiability problem is preserved across various fragments. For instance, if T is the theory of real numbers, the satisfiability in T of full-first order formulas is decidable. However, one can build increasingly faster solvers by restricting the language respectively to quantifier-free formulas, linear equations and inequations, difference equations, inequations between variables, and so on [?].

Certain pairs of theories and input languages are very common in the field, and are often conveniently considered as a single entity. In recognition of this practice, the SMT-LIB format allows one to pair together a background theory and an input language into a *sublogic*, or, more simply, *logic*. We call these pairs (sub)logics because, intuitively, each of them defines a sublogic of SMT-LIB's underlying logic for restricting both the set of allowed models—to the models of the background theory—and the set of allowed formulas—to the formulas in the input language.

---

[3] By "formal" here we will always mean written in a machine-readable and processable format, as opposed to written in free text, no matter how rigorously.

[4] By efficiency here we do not necessarily refer to worst-case time complexity, but to efficiency in practice.

3 The SMT-LIB Underlying Logic and Language

Under the SMT-LIB translation semantics, from a semantic viewpoint the ultimate underlying logic for SMT-LIB is $FOL^=$, the classical (unsorted) first-order logic with equality. The *external* underlying logic and its language are, however, many-sorted.

As a typed language, the SMT-LIB language for formulas and theories is intentionally limited in expressive power. In essence, the language allows one to declare sorts (types) only by means of sort symbols, to specify the interface, or *rank* of function and predicate symbols in terms of the declared sorts, and to specify the sort of quantified variables.

In type theory terms, the language has no subtypes, no type constructors, no type quantifiers, no provisions for parametric or subsort polymorphism, and so on. Only explicit (ad-hoc) overloading of function or predicate symbols—by which a symbol could have more than one rank—is allowed. The idea is to provide, at least in this version, just enough expressive power to represent typical benchmarks without getting bogged down by the complexity of a more sophisticated type system.

3.1 The Formula Sublanguage

The syntax of formulas in the SMT-LIB language extends the standard abstract syntax of $FOL^=$ with a construct for declaring the sort of quantified variables, plus the following non-standard constructs:

- an **if-then-else** construct for terms,

- an **if-then-else** logical connective,

- a **let** construct for terms,

- a **let** construct for formulas, and

- a **distinct** construct for declaring a number of values as pairwise distinct.

- an annotation mechanism for terms and formulas.

Except for the first extension, needed to support sorts, the other extensions are provided for greater convenience. We discuss each of them in the following.

An abstract grammar and syntax for a superset of the SMT-LIB language of logical formulas is defined in Figure 1 by means of production rules. This is a *superset* of the language because it contains ill sorted terms and formulas. The proper well sorted subset is defined later by means of a separate sort system.

The rules assume as given the following sets of symbols:

(Annotations)	α	::=	$a = v$

(Terms)	u	::=	$x \mid n \mid r \mid f\,t^* \mid \mathbf{ite}\ \varphi\ t_1\ t_2$

(Annot. Terms)	t	::=	$u\,\alpha^*$

(Connectives)	κ	::=	$\mathbf{not} \mid \mathbf{implies} \mid \mathbf{and} \mid \mathbf{or} \mid \mathbf{xor} \mid \mathbf{iff}$

(Atoms)	A	::=	$\mathbf{true} \mid \mathbf{false} \mid \xi \mid p\,t^* \mid\ =t^* \mid \mathbf{distinct}\ t^*$

$$
\begin{aligned}
\text{(Formulas)} \quad \psi \ ::=\ & A \mid \kappa\,\varphi^+ \mid \exists\,(x{:}s)^+\ \varphi_0 \mid \forall\,(x{:}s)^+\ \varphi_0 \\
\mid\ & \mathbf{let}\ x = t\ \mathbf{in}\ \varphi_0 \mid \mathbf{flet}\ \xi = \varphi_0\ \mathbf{in}\ \varphi_1 \\
\mid\ & \mathbf{if}\ \varphi_0\ \mathbf{then}\ \varphi_1\ \mathbf{else}\ \varphi_2
\end{aligned}
$$

(Annot. Formulas)	φ	::=	$\psi\,\alpha^*$

$x \in \mathcal{X},$ the set of term variables $\qquad \xi \in \Xi,$ the set of formula variables
$n \in \mathcal{N},$ the set of numerals $\qquad r \in \mathcal{R},$ the set of positive rationals
$f \in \mathcal{F},$ the set of function symbols $\quad p \in \mathcal{P},$ the set of predicate symbols
$s \in \mathcal{S},$ the set of sort symbols $\qquad a \in \mathcal{A},$ the set of attributes
$v \in \mathcal{V},$ the set of attribute values

Figure 1: Abstract syntax for unsorted terms and formulas

- an infinite set $\mathcal{X}$ of *term variables*, standard first-order variables that can be used in place of a term;

- an infinite set Ξ of *formula variables*, second-order variables that can be used in place of a formula;

- the infinite set $\mathcal{N}$ of all *numerals* $(0, 1, 2, \ldots)$, for the natural numbers;

- the infinite set $\mathcal{R}$ for all the *positive rationals* (intuitively, all the value denoted by fractions $\frac{m}{n}$);

- an infinite set $\mathcal{F}$ of *function symbols*;

- an infinite set $\mathcal{P}$ of *predicate symbols*;

- an infinite set $\mathcal{S}$ of *sort symbols*;

- an infinite set $\mathcal{A}$ of *attribute symbols*;

- a set $\mathcal{V}$ of attribute *values*.

It is required that $\mathcal{A}$ be disjoint from all the other sets and that $\mathcal{N}$ and $\mathcal{R}$ be pairwise disjoint. The remaining sets need not be disjoint because the syntax of terms and formula allows one to infer to which set a particular symbol/value belongs.

In Figure 1 and later in the paper, boldface words denote terminal symbols, parentheses are meta-symbols, used just for grouping—they are not symbols of the abstract language. The meta-operator $(\_)^*$ denotes as usual zero or more repetitions of its argument, while the meta-operator $(\_)^+$ denotes one or more repetitions of its argument. Function/predicate applications are denoted simply by juxtaposition, as this is enough at the abstract level.

In the production rules, the letter a denotes attribute symbols, the letter v attribute values, the letter x term variables, the letter ξ formula variables, the letter n numerals, the letter r rational constants, the letter f functions symbols, the letter p predicate symbols, the letter u terms, the letter t annotated terms, the letter s sort symbols, the letter ψ formulas, and the letter φ annotated formulas.

The given grammar does not distinguish between constant and function symbols (they are all defined as members of the set $\mathcal{F}$), and between propositional variables and predicate symbols (they are all defined as members of the set $\mathcal{P}$). These distinctions are really a matter of arity, which is taken care of later by the well-sortedness rules. A similar observation applies to the logical connectives, the members of κ class, and the number of arguments they are allowed take.

From now on, we will simply say *term* to mean a possibly annotated term, and *formula* to mean a possibly annotated formula.

As usual, one can speak of the *free variables* of a formula. Formally, The set $\mathcal{V}ar(t)$ of free term variables in a term t and the set $\mathcal{V}ar(\varphi)$ of free term variables in a formula φ are respectively defined below by structural induction.

$$
\begin{aligned}
\mathcal{V}ar(x) &= \{x\} \\
\mathcal{V}ar(n) &= \emptyset \\
\mathcal{V}ar(r) &= \emptyset \\
\mathcal{V}ar(f \; t_1 \cdots t_k) &= \mathcal{V}ar(t_1) \cup \cdots \cup \mathcal{V}ar(t_k) \\
\mathcal{V}ar(\mathbf{ite} \; \varphi_0 \; t_1 \; t_2) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(t_1) \cup \mathcal{V}ar(t_2) \\
\mathcal{V}ar(u \; \alpha_1 \cdots \alpha_k) &= \mathcal{V}ar(u)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}ar(\mathbf{true}) &= \emptyset \\
\mathcal{V}ar(\mathbf{false}) &= \emptyset \\
\mathcal{V}ar(\xi) &= \emptyset \\
\mathcal{V}ar(\pi \; t_1 \cdots t_k) &= \mathcal{V}ar(t_1) \cup \cdots \cup \mathcal{V}ar(t_k) \\
& \quad \text{if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \\
\mathcal{V}ar(\kappa \; \varphi_1 \cdots \varphi_k) &= \mathcal{V}ar(\varphi_1) \cup \cdots \cup \mathcal{V}ar(\varphi_k) \\
\mathcal{V}ar(\forall \; x_1{:}s_1 \ldots x_k{:}s_k \; \varphi_0) &= \mathcal{V}ar(\varphi_0) \setminus \{x_1 \ldots x_k\} \\
\mathcal{V}ar(\exists \; x_1{:}s_1 \ldots x_k{:}s_k \; \varphi_0) &= \mathcal{V}ar(\varphi_0) \setminus \{x_1 \ldots x_k\} \\
\mathcal{V}ar(\mathbf{let} \; x = t \; \mathbf{in} \; \varphi_0) &= \mathcal{V}ar(t) \cup (\mathcal{V}ar(\varphi_0) \setminus \{x\}) \\
\mathcal{V}ar(\mathbf{flet} \; \xi = \varphi_0 \; \mathbf{in} \; \varphi_1) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(\varphi_1) \\
\mathcal{V}ar(\mathbf{if} \; \varphi_0 \; \mathbf{then} \; \varphi_1 \; \mathbf{else} \; \varphi_2) &= \mathcal{V}ar(\varphi_0) \cup \mathcal{V}ar(\varphi_1) \cup \mathcal{V}ar(\varphi_2) \\
\mathcal{V}ar(\psi \; \alpha_1 \cdots \alpha_k) &= \mathcal{V}ar(\psi)
\end{aligned}
$$

The notions of *free/bound occurrence* of a variable and of *scope* of a variable's occurrence are defined as usual.[5] Note that, also as usual, in the expression $\mathbf{let} \; x = t \; \mathbf{in} \; \varphi_0$, occurrences of the variable x are bound in φ_0 but not in t.

The set $\mathcal{F}ar(\varphi)$ of *free formula variables* in a formula φ is analogously defined as follows:

$$
\begin{aligned}
\mathcal{F}ar(\mathbf{true}) &= \emptyset \\
\mathcal{F}ar(\mathbf{false}) &= \emptyset \\
\mathcal{F}ar(\xi) &= \{\xi\} \\
\mathcal{F}ar(\pi \; t_1 \cdots t_k) &= \emptyset \;\; \text{if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \\
\mathcal{F}ar(\kappa \; \varphi_1 \cdots \varphi_k) &= \mathcal{F}ar(\varphi_1) \cup \cdots \cup \mathcal{F}ar(\varphi_k) \\
\mathcal{F}ar(\forall \; x_1{:}s_1 \ldots x_k{:}s_k \; \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\exists \; x_1{:}s_1 \ldots x_k{:}s_k \; \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\mathbf{let} \; x = t \; \mathbf{in} \; \varphi_0) &= \mathcal{F}ar(\varphi_0) \\
\mathcal{F}ar(\mathbf{flet} \; \xi = \varphi_0 \; \mathbf{in} \; \varphi_1) &= \mathcal{F}ar(\varphi_0) \cup (\mathcal{F}ar(\varphi_1) \setminus \{\xi\}) \\
\mathcal{F}ar(\mathbf{if} \; \varphi_0 \; \mathbf{then} \; \varphi_1 \; \mathbf{else} \; \varphi_2) &= \mathcal{F}ar(\varphi_0) \cup \mathcal{F}ar(\varphi_1) \cup \mathcal{F}ar(\varphi_2) \\
\mathcal{F}ar(\psi \; \alpha_1 \cdots \alpha_k) &= \mathcal{F}ar(\psi)
\end{aligned}
$$

---

[5] See any standard text book on first-order logic.

In practice, the only operator that binds formula variables is **flet**. Similarly to **let** for variables, in the expression **flet** $\xi = \varphi_0$ **in** φ_1, occurrences of ξ are bound in φ_1 but not in φ_0.

We call a formula φ *closed* iff $\mathcal{V}ar(\varphi) = \mathcal{F}ar(\varphi) = \emptyset$.

3.2 The additional constructs

The following constructs are not usually found in standard descriptions of many-sorted logic. While they do not extend the power of SMT-LIB's underlying logic beyond first order, they are convenient for representing SMT benchmarks.

The *if-then-else* construct for terms

This construct is very common in benchmarks coming from hardware verification. Technically, it is a function symbol of arity 3, taking as first argument a formula and as second and third arguments a term. Semantically, an expression like

$$\mathbf{ite}(\varphi, t_1, t_2)$$

evaluates to the value of t_1 in every interpretation that makes φ true, and to the value of t_2 in every interpretation that makes φ false.

Although it can be defined in terms of more basic constructs, this construct provides important structural information that a solver may be able to use advantageously. More importantly, for certain applications the use of **ite** constructs can result in an exponential reduction in the size of a benchmark—with respect to an equivalent reformulation of the benchmark as a formula without **ite**s. For these reasons, the SMT-LIB format supports them natively.

The *if-then-else* logical connective

This construct is also common in verification benchmarks. It is used to build a formula of the form

$$\text{if } \varphi_0 \text{ then } \varphi_1 \text{ else } \varphi_2$$

which is semantically equivalent to the formula

$$(\varphi_0 \textbf{ implies } \varphi_1) \textbf{ and } (\textbf{not } \varphi_0 \textbf{ implies } \varphi_1)$$

The SMT-LIB format supports this *if-then-else* connective natively for the same reasons it supports the **ite** term constructor.

The *let* construct for terms

This construct builds a formula of the form

$$\textbf{let } x = t \textbf{ in } \varphi_0$$

which is semantically equivalent to the formula obtained from φ_0 by simultaneously replacing every free occurrence of x in φ_0 by t, and renaming as necessary the bound variables of φ_0 so that the variables of t remain free.

The construct is convenient for benchmark compactness as it allows one to replace multiple occurrences of the same term by a variable. It is also useful for a solver because it saves the solver the effort of recognizing the various occurrences of the same term as such.

The *let* construct for formulas

This construct builds a formula of the form

$$\textbf{flet } \xi = \varphi_0 \textbf{ in } \varphi_1$$

which is semantically equivalent to the formula obtained from φ_1 by simultaneously replacing every free occurrence of ξ in φ_1 by φ_0, and renaming as necessary the bound variables of φ_1 so that the free variables of φ_0 remain free.

The rationale for supporting this construct in SMT-LIB is similar to that for supporting the let construct for terms.

The *distinct* construct

This construct is also supported for conciseness. It is a variadic construct for building formulas of the form
$$\textbf{distinct}(t_1, \ldots, t_n)$$
with $n \geq 2$. Semantically, it is equivalent to the conjunction of all disequations of the form $\textbf{not}(t_i = t_j)$ for $1 \leq i < j \leq n$.

The annotation mechanism for terms and formulas

Each term or formula can be annotated with a list of *attribute-value* pairs of the form
$$a = v$$
where a is an attribute's name and v is the attribute's value.

The syntax of attribute values is user-dependent and is therefore left unspecified by the SMT-LIB format.

Annotations are meant to provide extra-logical information which, while not changing the semantics of a term or formula, may be useful to a theory solver. It is expected that typical annotations will provide operational information for the solver. For instance, the annotation α in a formula of the form

$$\forall\, x_1{:}s_1 \ldots x{:}s_k\ \varphi_0\ \alpha$$

might specify an instantiation pattern for the quantifier $\forall\, x_1{:}s_1 \ldots x{:}s_k$, as done in the Simplify prover [?]. Or, for formulas that represent verification conditions for a program, the annotation might contain information relating the formula to the original code the formula was derived from.[6]

3.3   Well-sorted formulas

The SMT-LIB language of formulas is the largest set of well-sorted formulas contained in the language generated by the production rules for annotated formulas in the grammar of Figure 1.

Well-sorted formulas are defined by means of a set of sorting rules, similar in format and spirit to the kind of typing rules found in the programming languages literature. The rules are based on the following definition of a (many-sorted) signature.

Definition 1 (SMT-LIB Signature) An *SMT-LIB signature* Σ is a tuple consisting of:

- a non-empty set $\Sigma^{\mathrm{S}} \subseteq \mathcal{S}$ of sort symbols, a set $\Sigma^{\mathrm{F}} \subseteq \mathcal{F}$ of function symbols, a set $\Sigma^{\mathrm{P}} \subseteq \mathcal{P}$ of predicate symbols;

- a total mapping from the term variables $\mathcal{X}$ to Σ^{S};

- a total relation from Σ^{F} to $(\Sigma^{\mathrm{S}})^{+}$, the non-empty sequences of elements of Σ^{S}, never containing two pairs of the form $(f, s_1 \cdots s_n s)$ and $(f, s_1 \cdots s_n s')$ with s and s' different sorts;

- a total relation from Σ^{P} to $(\Sigma^{\mathrm{S}})^{*}$, the sequences of elements of Σ^{S}.

The sequence of sorts associated by Σ to a function/predicate symbol is called the *rank* of the symbol.

As usual, the rank of a function symbol specifies the expected sort of the symbol's argument and result. Similarly for predicate symbols with the difference that it is possible to associate a predicate symbol to the empty sequence, denoting that the

---

[6] A similar idea is used in the ESC/Java system with the use of a special "label" predicate [?].

$$\frac{}{\Sigma \vdash_{\mathrm{t}} \; x : s} \quad \text{if } x : s \in \Sigma \qquad\qquad \frac{}{\Sigma \vdash_{\mathrm{t}} \; n : s} \quad \text{if } n : s \in \Sigma$$

$$\frac{}{\Sigma \vdash_{\mathrm{t}} \; r : s} \quad \text{if } r : s \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{t}} \; t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} \; t_k : s_k}{\Sigma \vdash_{\mathrm{t}} \; f \; t_1 \; \cdots \; t_k : s_{k+1}} \quad \text{if } f : s_1 \cdots s_{k+1} \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \; \varphi \qquad \Sigma \vdash_{\mathrm{t}} \; t_1 : s \qquad \Sigma \vdash_{\mathrm{t}} \; t_2 : s}{\Sigma \vdash_{\mathrm{t}} \; \mathbf{ite} \; \varphi \; t_1 \, t_2 : s}$$

Figure 2: Well-sortedness rules for terms

symbol is a propositional predicate. Note that the sets Σ^{S}, Σ^{F} and Σ^{P} of an SMT-LIB signature are not required to be disjoint. So it is possible for a symbol to be both a function and a predicate symbol, say. This causes no ambiguity in the language because positional information is enough to determine during parsing if a given occurrence of a symbol is a function, predicate or sort symbol. Also, it is possible for a function or predicate symbol to have more than one rank. In other words, *ad hoc* overloading of function/predicate symbols is allowed. This lifts a no-overloading restriction in Version 1.1, which is too strong for certain theories such as, for instance, some popular theories of bit vectors. The requirement in Definition 1 that every variable has a unique sort and that no function symbol has distinct ranks of the form $s_1 \cdots s_n s$ and $s_1 \cdots s_n s'$ guarantees that every well-sorted term has a *unique* sort.

Figure 2 provides a set of rules defining well-sorted terms, while Figure 3 provides a rule set defining well-sorted formulas. The sort rules presuppose the existence of an SMT-LIB signature Σ. Strictly speaking then, the SMT-LIB language is a family of languages parametrized by Σ. As explained later, for each benchmark φ and theory T, the specific signature is jointly defined by the specification of T and that of the benchmark containing φ.

The format and the meaning of the sort rules in the two figures is pretty standard and should be largely self-explanatory. The integer index k in the rules is assumed ≥ 0; the notation $x{:}s \in \Sigma$ means that Σ maps the variable x to the sort s. The notation $f{:}s_1 \cdots s_{n+1} \in \Sigma$ means that $f \in \Sigma^{\mathrm{F}}$ and Σ associates f to the sort sequence $s_1 \cdots s_{n+1}$ (and similarly for number and predicate symbols). The expression $\Sigma, x{:}s$ denotes the signature that maps x to the sort s and otherwise coincides with Σ.

A term t generated by the grammar in Figure 1 is *well-sorted (with respect to Σ)* if the expression $\Sigma \vdash_{\mathrm{t}} \; t{:}s$ is derivable by the sort rules in Figure 2 for some

15

$$\overline{\Sigma \vdash_f \textbf{true}} \qquad\qquad \overline{\Sigma \vdash_f \textbf{false}} \qquad\qquad \overline{\Sigma \vdash_f \xi}$$

$$\frac{\Sigma \vdash_t t_1 : s_1 \quad \cdots \quad \Sigma \vdash_t t_k : s_k}{\Sigma \vdash_f p\ t_1\ \cdots\ t_k} \quad \text{if } p : s_1 \cdots s_k \in \Sigma$$

$$\frac{\Sigma \vdash_t t_1 : s \quad \cdots \quad \Sigma \vdash_t t_{k+2} : s}{\Sigma \vdash_f =\ t_1\ \cdots\ t_{k+2}} \qquad \frac{\Sigma \vdash_t t_1 : s \quad \cdots \quad \Sigma \vdash_t t_{k+2} : s}{\Sigma \vdash_f \textbf{distinct}\ t_1\ \cdots\ t_{k+2}}$$

$$\frac{\Sigma \vdash_f \varphi}{\Sigma \vdash_f \textbf{not}\ \varphi} \qquad \frac{\Sigma \vdash_f \varphi_1 \quad \Sigma \vdash_f \varphi_2}{\Sigma \vdash_f \textbf{impl}\ \varphi_1\ \varphi_2} \qquad \frac{\Sigma \vdash_f \varphi_0 \quad \Sigma \vdash_f \varphi_1 \quad \Sigma \vdash_f \varphi_2}{\Sigma \vdash_f \textbf{if}\ \varphi_0\ \textbf{then}\ \varphi_1\ \textbf{else}\ \varphi_2}$$

$$\frac{\Sigma \vdash_f \varphi_1 \quad \cdots \quad \Sigma \vdash_f \varphi_{k+2}}{\Sigma \vdash_f c\ \varphi_1\ \cdots\ \varphi_{k+2}} \quad \text{if } c \in \{\textbf{and}, \textbf{or}, \textbf{xor}, \textbf{iff}\}$$

$$\frac{\Sigma,\ x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_f \varphi}{\Sigma \vdash_f \exists x_1{:}s_1 \ldots x{:}s_{k+1}\ \varphi} \qquad \frac{\Sigma,\ x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_f \varphi}{\Sigma \vdash_f \forall x_1{:}s_1 \ldots x{:}s_{k+1}\ \varphi}$$

$$\frac{\Sigma \vdash_t t : s \quad \Sigma, x : s \vdash_f \varphi}{\Sigma \vdash_f \textbf{let}\ x\ =t\ \textbf{in}\ \varphi} \qquad \frac{\Sigma \vdash_f \varphi_0 \quad \Sigma \vdash_f \varphi_1}{\Sigma \vdash_f \textbf{flet}\ \xi\ =\varphi_0\ \textbf{in}\ \varphi_1}$$

Figure 3: Well-sortedness rules for formulas

sort $s \in \Sigma^{\mathrm{S}}$. In that case, we say that t *has sort* s. It is possible to show that, as mentioned earlier, a term as at most one sort.

A formula φ generated by the grammar in Figure 1 is *well-sorted (with respect to Σ)* if the expression $\Sigma \vdash_{\mathrm{f}} \varphi$ is derivable by the sort rules in Figure 3.

Definition 2 (SMT-LIB formulas) The SMT-LIB language for formulas is the set of all closed well-formed formulas.

Note that the SMT-LIB language for formulas contains only closed formulas. This is mostly a technical restriction, motivated by considerations of convenience. In fact, with a closed formula φ of a signature Σ the particular mapping of term variables to sorts defined by Σ is irrelevant. The reason is that the formula itself contains its own sort declaration for its term variables, either explicitly, for the variables bound by a quantifier, or implicitly, for the variables bound by a **let**. Using only closed formulas then simplifies the task of specifying their signature, as it becomes unnecessary to specify how the signature maps the elements of $\mathcal{X}$ to the signature's sorts.

There is no loss of generality in allowing only closed formulas because, as far as satisfiability of formulas is concerned, every formula φ with free variables $\mathcal{V}ar(\varphi) = \{x_1, \ldots, x_n\}$, where each x_i is expected to have sort s_i, can be rewritten as

$$\exists\, x_1{:}s_1 \ldots x_n{:}s_n\ \varphi.$$

An alternative way to avoid free variables in benchmarks is defined in Section 5.

A similar situation arises with formula variables. In fact, all free occurrences of a formula variable can be replaced by a fresh predicate symbol p declared in Σ as having empty arity.

4 The SMT-LIB Theory Language

This version of the format considers only the specification of basic background theories. Facilities for specifying structured theories will be introduced in a later version.

A *theory declaration* defines both a many-sorted signature for a theory and the theory itself. The abstract syntax for a theory declaration in the SMT-LIB format is provided in Figure 4. The syntax follows an attribute-value-based format.

In addition to the already defined symbols and syntactical categories, the production rules in Figure 4 assume as given the following sets of symbols:

- an infinite set $\mathcal{W}$ of *character strings*, meant to contain free text;

- an infinite set $\mathcal{T}$ of *theory names*, used to give a name to each theory.

$$
\begin{array}{llll}
\text{(Fun. sym. declaration)} & D_f & ::= & f\ s^+\ \alpha \\[2mm]
\text{(Pred. sym. declaration)} & D_p & ::= & p\ s^*\ \alpha \\[2mm]
\text{(Attribute declaration)} & P_T & ::= & \mathbf{sorts} = s^+ \\
& & | & \mathbf{funs} = (D_f)^+ \ \ | \ \ \mathbf{preds} = (D_p)^+ \\
& & | & \mathbf{definition} = w \ \ | \ \ \mathbf{axioms} = \varphi^+ \\
& & | & \mathbf{notes} = w \ \ | \ \ \alpha \\[2mm]
\text{(Theory declaration)} & D_T & ::= & \mathbf{theory}\ T\ \mathbf{begin} \\
& & & \quad (P_T)^+ \\
& & & \mathbf{end}
\end{array}
$$

$$T \in \mathcal{T}, \quad \text{the set of theory names} \qquad w \in \mathcal{W}, \quad \text{the set of character strings}$$

Figure 4: Abstract syntax for theories

In the rules, the letter w denotes strings and the letter T theory names.

The symbols **funs**, **preds**, **axioms**, **notes**, **sorts**, and **definition** are reserved attribute symbols from $\mathcal{A}$. Their sets of values are as specified in the rules. In addition to the predefined attributes, a theory declaration D_T can contain an unspecified number of user defined attributes, and their values—formalized in the grammar simply as annotations α.

The rationale for allowing user-defined attributes is the same as in other attribute-value-based language (such as, e.g., BibTeX). It makes the SMT-LIB format more flexible and customizable. The understanding is that user-defined attributes are allowed but need not be supported by an SMT solver for the solver to be considered *SMT-LIB compliant*. We expect however that continued use of the SMT-LIB format will make certain user-defined attributes widely used. Those attributes might then be officially introduced in the format (as non-user-defined attributes) in later versions.

Note that in the abstract syntax of a theory declaration, attribute-value pairs can appear in any order. However, they are subject to the restrictions below.

Definition 3 (Theory Declarations) *The only legal theory declarations in the SMT-LIB format are those that satisfy the following restrictions.*

1. *They include a declaration of the attributes* **sorts** *and* **definition**[7].

2. *They contain at most one declaration per attribute.*

---

[7] Which makes those attributes non-optional.

3. *There are no two declarations of the form $f\ s_1 \cdots s_n s$ and $f\ s_1 \cdots s_n s'$ for the same function symbol f with the arity sequences differing only for the last sort.*

4. *All sorts in function/predicate symbol declarations are listed in the **sort** attribute declaration.*

5. *The definition of the theory, however provided in the **definition** attribute, refers only to the declared sort, function and predicate symbols.*

6. *The formulas in the **axioms** attribute are built over the symbols declared in the **sort**, **funs** and **pred** attributes.*

The first restriction is explained in the following. The second restriction is just to simplify later the definition of a declaration semantics. The third restriction is needed for terms to have at most one sort.

Some attributes, such as **definition** for instance, are *informal attributes* in the sense that their value (w) is free text. Ideally, a formal specification of the given free-text attributes would be preferable to free text in order to avoid ambiguities and misinterpretation. The choice of using free text for these attributes is motivated by practicality reasons. In fact, (i) these attributes are meant to be read by human readers, not programs, and (ii) the amount of effort needed to first devise a formal language for these attributes and then specify their values for each theory in the library does not seem justified by the current goals of SMT-LIB.

The signature of a theory is defined by the attributes **sorts**, **funs** and **preds** in the obvious way. A declaration D_f for a function symbol f specifies the symbol's rank, and may contain additional, user-defined annotations. A typical annotation might specify that f is associative, say. While this property is expected to be specified also in the definition of the theory (or to be a consequence thereof), an explicit declaration at this point could be useful for certain solvers that treat associative symbols in a special way. Similarly, a declaration D_p for a predicate symbol p specifies the symbol's rank, and may be augmented with user-defined annotations.

This version of the format does not specify any predefined annotations for function and predicate symbols. Future versions might do so, depending on the recommendations and the feedback of the SMT-LIB user community.

The **funs** and **preds** attributes are optional because a theory might lack function or predicate symbols. The **sorts** attribute, however, is not optional because sorted frameworks require the existence of at least one sort. This is no real limitation of course because, for instance, any unsorted theory can be always seen as at least one-sorted.

The non-optional **definition** attribute is meant to contain a natural language definition of the theory. While this definition is expected to be as rigorous as possible, it does not have to be a formal one. Some theories (like the theory of real numbers) are well known, and so just a reference to their common name might be enough. For

$$
\begin{array}{lll}
\text{(Attribute declaration)} & P_L & ::= \quad \textbf{theory} = T \ \mid \ \textbf{language} = w \\
& & \quad\quad \mid \quad \textbf{extensions} = w \ \mid \ \textbf{notes} = w \ \mid \ \alpha \\
\\
\text{(Logic declaration)} & D_L & ::= \quad \textbf{logic } L \textbf{ begin} \\
& & \quad\quad\quad (P_L)^+ \\
& & \quad\quad \textbf{end}
\end{array}
$$

$$
L \in \mathcal{L}, \quad \text{the set of logic names}
$$

Figure 5: Abstract syntax for logics

theories that have a small set of axioms (or axiom schemas), it might be convenient to list the actual axioms. For some other theories, a mix of formal notation and informal explanation might be more appropriate.

Formal, first-order axioms that define part of or a whole theory can be additionally provided in the optional **axioms** attribute as a list of SMT-LIB formulas. In addition to being more rigorous, the formal definition of (a part of) a theory provided in the **axioms** attribute, is useful for solvers that might not have (that part of) the theory built in, but accept theory axioms input. In essence, this is currently the case for instance for the solvers Simplify [?], CVC Lite [?], haRVey [?], and Argo-lib [?].

The optional attribute **notes** is meant to contain documentation information such as authors, date, version, etc. of a specification, although this information could also be provided by more specific user-defined attributes.

4.1 SMT-LIB Logics

The SMT-LIB format allows the explicit definition of sublogics of its main logic—many-sorted first-order logic with equality—that restrict both the main logic's syntax and semantics. A new (sub)logic is defined in the SMT-LIB language by a *logic declaration* whose abstract syntax is provided in Figure 5.

In addition to the already defined symbols and syntactical categories, the production rules in Figure 4 assume as given an infinite set $\mathcal{L}$ of *logic names*, used to give a name to each logic. In the rules, the letter L denotes logic names.

The symbols **theory**, **language**, **extensions**, and **notes** are reserved attribute symbols from $\mathcal{A}$. Their sets of values are as specified in the rules. As for theories, a logic declaration D_L can contain an unspecified number of user defined attributes, and their values—formalized in the grammar simply as annotations α.

Definition 4 (Logic Declarations) *The only legal logic declarations in the SMT-LIB format are those that satisfy the following restrictions:*

1. *They include a declaration of the attributes* **theory** *and* **language**.

2. *They contain at most one declaration per attribute.*

3. *The value of the attribute* **theory** *coincides with the name of a theory T for some theory declaration D_T in SMT-LIB;*

The text attribute **notes** serves the same purpose as in theory declarations.

The attribute **theory** refers to a theory T in SMT-LIB. As we will see, the effect of this attribute is to declare that the logic's sort, function and predicate symbols are restricted to those in the signature of T, and that the logic's semantics is restricted to the models of T.

The attribute **language** describes in free text the logic's *language*, that is, the specific subset of SMT-LIB formulas that are allowed in the logic. As explained in Section 2.3, this information is useful for tailoring solvers to the specific sublanguage of formulas used in a set of benchmarks. The attribute is text valued because it has mostly documentation purposes for the benefit of benchmark users. A natural language description of the logic's language seems therefore adequate for this purpose. However, we expect that the language will be at least partially specified in some formal fashion in this attribute, for instance by using BNF rules.

It is understood that the formulas in the logic's language are built over the signature Σ of the associated theory unless this signature is explicitly expanded in the text of attribute **language** with new[8] symbols. In that case, *the text must explicitly indicate which new sort, function or predicate symbols are included in the expansion.* See later for why it is convenient to expand the language with free symbols. In effect then, an SMT-LIB logic's language is really parametric in such a signature expansion.

The optional attribute **extensions** is meant to document any notational conventions, or syntactic sugar, allowed in formulas of this logic—and hence in benchmarks for the logic. This is useful because in common practice the syntax of a logic is often extended for convenience with syntactic sugar.

One example of a logic declaration that uses the attributes **theory**, **language** and **extensions** is what we could call *linear Presburger arithmetic*. In this logic, the theory T is the theory of natural numbers with signature $\{0, s, +, <\}$ (s for the successor function) as axiomatized by Presburger [**?**]. The language is the set of Boolean combinations of atomic formulas over this signature expanded with an infinite number of free constants. Examples of syntactic conventions defined in the **extension** attribute could be the use of a numeral n for the n-fold application of s to 0, and the use of the expression $n * t$ for the term $\underbrace{t + \cdots + t}_{n \text{ times}}$.

---

[8] That is, *uninterpreted*, or *free*, in logic parlance.

(Formula status)	σ	$::=$	**sat** | **unsat** | **unknown**
(Attribute-value pair)	P_b	$::=$	**logic** $= L$ | **formula** $= \varphi$ | **status** $= \sigma$
		|	**assumption** $= \varphi$ | **extrasorts** $= s^+$
		|	**extrafuns** $= (D_f)^+$ | **extrapreds** $= (D_p)^+$
		|	**notes** $= w$ | α
(Benchmark declaration)	D_b	$::=$	**benchmark** b **begin**
			$\quad (P_b)^+$
			end

$$b \in \mathcal{B}, \quad \text{the set of benchmark names}$$

Figure 6: Abstract syntax for benchmarks

Note that function or predicate symbols introduced as syntactic sugar are not formally defined in the **funs** or **pred** symbols, or anywhere else. In a sense, syntactic sugar does not officially exist, even if it is allowed in benchmarks for convenience. This deficiency of the SMT-LIB language, which is meant to be resolved in future versions, is mostly due to the impossibility of (formally) defining an infinite signature at the moment. That is why, for instance, in the example above numerals are introduced as syntactic sugar for ground terms over $\{s, 0\}$.

5 The SMT-LIB Benchmark Language

In SMT-LIB, a benchmark is a closed SMT-LIB formula with attached additional information, specified in a *benchmark* declaration. In addition to the formula itself, a *benchmark* declaration contains a reference to its background logic, and an optional specification of additional sort, function and predicate symbols.

The abstract syntax for a benchmark declaration in the SMT-LIB format is provided in Figure 6. In addition to the already defined symbols and syntactical categories, the production rules in the figure assume as given an infinite set $\mathcal{B}$ of *benchmark names*, used to give a name to each benchmark. In the rules, the letter b denotes benchmark names.

The symbols **assumption**, **formula**, **status**, **logic**, **extrasorts**, **extrafuns**, **extrapreds**, and **notes** are reserved attribute symbols from $\mathcal{A}$. Their sets of values are as specified in the rules. As before, benchmark declarations can also contain user-defined attributes and their values—formalized again as annotations α.

Definition 5 (Benchmark Declarations) *The only legal benchmark declarations in the SMT-LIB format are those that satisfy the following restrictions.*

1. *They contain exactly one declaration of the attributes* **logic**, **formula** *and* **status**.

2. *The value of the attribute* **logic** *coincides with the name of a logic L for some logic declaration D_L in SMT-LIB.*

3. *Every symbol declared in the* **extrasorts**, **extrafuns**, *and* **extrapreds** *attributes must be part of the signature expansion defined in the* **language** *attribute of the logic L.*

4. *The declarations of the* **extrasort** *attribute, which can be more than one, collectively contain no more than one occurrence of the same sort symbol, and this symbol does not occur in the* **sort** *attribute of the theory declaration associated to D_L.*

5. *The total set of function symbols in* **extrafuns** *declarations, which can be more than one, and in the* **funs** *attribute of the theory declaration associated to D_L satisfy Requirement 3 in Definition 3.*

6. *The sort symbols occurring in the* **extrafuns** *or* **extrapreds** *declarations are either declared in* **extrasorts** *or belong to the signature of the logic L;*

7. *The formulas in the* **assumption** *or* **formula** *attributes are in the language of L, and their free symbols are only among those declared in the attributes* **extrasorts**, **extrafuns** *and* **extrapreds**.

In a benchmark declaration of the form:

$$
\begin{aligned}
&\textbf{benchmark } b \textbf{ begin} \\
&\quad \textbf{logic} = L \\
&\quad \textbf{assumption} = \varphi_1 \\
&\qquad\qquad \vdots \\
&\quad \textbf{assumption} = \varphi_n \\
&\quad \textbf{formula} = \varphi \\
&\quad \textbf{status} = \sigma \\
&\textbf{end}
\end{aligned}
$$

with $n \geq 0$, the formulas $\varphi_1, \ldots, \varphi_n$ and φ together constitute the benchmark. For ease of reference, let us refer to L as the background logic for the benchmark, and to the theory contained in L's declaration as the background theory. The intended test

is whether the formula φ is satisfiable in the background logic *under the assumptions* $\varphi_1, \ldots, \varphi_n$; in other words, whether the formula $(\textbf{and } \varphi_1 \cdots \varphi_n \ \varphi)$ is satisfiable in the background theory. Clearly, a benchmark with assumptions $\varphi_1, \ldots, \varphi_n$ and main formula φ could also be given as the semantically equivalent formula $(\textbf{and } \varphi_1 \cdots \varphi_n \ \varphi)$ with no assumptions. The rationale for the **assumption** attribute is entirely operational. Many SMT solvers can process assumptions more efficiently if they are explicitly identified as such, as opposed to given together with the *query* formula φ. We call $(\textbf{and } \varphi_1 \cdots \varphi_n \ \varphi)$ the formula of the benchmark.

The attribute **status** of a benchmark declaration records whether the benchmark's formula is known to be (un)satisfiable in the associated background theory. Knowing about the satisfiability of a benchmark is useful for debugging solvers based on sound and complete procedures, or for evaluating the accuracy of solvers based on unsound or incomplete procedures.

The **extrasorts** attribute complements the **sorts** attribute of the background theory's declaration by declaring additional sort symbols. The **extrafuns** attribute complements the **funs** attribute of the theory declaration with additional function symbols with their rank. This includes the possibility of overloading an old function symbol with a new rank—provided that this overloading preserves sort uniqueness for well-sorted terms. The **extrapreds** attribute has a similar purpose, but for predicate symbols. In contrast with the symbols defined in the **extensions** attribute of a logic declaration, which are defined in terms of the symbols in the background theory, the symbols in **extrasorts**, **extrafuns** and **extrapreds** are "uninterpreted" in the theory. See Section 6 for a rigorous definition of what it means for a symbol to be uninterpreted in SMT-LIB.

Uninterpreted function or predicate symbols are found often in applications of satisfiability modulo theories, typically as a consequence of Skolemization or abstraction transformations applied to more complex formulas. Hence theory solvers typically accept formulas containing uninterpreted symbols in addition to the symbols of their background theory. The **extrasorts**, **extrafuns** and **extrapreds** attributes serve to declare any uninterpreted symbols occurring in the benchmark's formulas. The values of the **extrafuns**, **extrafuns** and **extrapreds** attributes are specified formally because, in effect, they dynamically expand the signature of the associated background theory, hence it is convenient for them to be directly readable by satisfiability procedures for that theory.

The **extrafuns** attribute is also useful for specifying benchmarks consisting of formulas with free term variables (such as quantifier-free formulas). As discussed in Section 3.3, the legal formulas of SMT-LIB do not contain free variables. One way to circumvent this restriction is to close formulas existentially. Another one is to replace each free term variable by a fresh constant symbol of the appropriate sort. In the second case, these extra constant symbols can be declared in the **extrafuns** attribute. A similar situation can occur in principle with free formula variables,

which can stand for unspecified predicates. Each free occurrence of a formula variable can be replaced by a fresh predicate symbol of empty arity (a.k.a, a propositional variable). Such symbols can be declared in the **extrapreds** attribute.

Note that all sort, function or predicate symbols occurring in benchmarks must be declared either in the background theory specification or in the **extrasorts**, **extrafuns** and **extrapreds** attributes. One could think of relaxing this restriction by adopting the convention that any undeclared function or predicate symbol occurring in a benchmark is automatically considered as uninterpreted. Contrary to unsorted logics, however, this approach is not feasible in SMT-LIB because it may not be possible in general to automatically infer (the sorts in) the rank of an undeclared symbol.

6   Semantics

In this section, we define precisely the notion of *satisfiability modulo theories* for SMT-LIB benchmarks. We do that in two alternative ways.

In the first, we associate with each benchmark a set of many-sorted structures (i.e., first-order models) and say that a benchmark is satisfiable modulo its background theory if its formula is satisfiable in at least one of these structures in the standard sense of many-sorted logic. In the second, we associate with each benchmark a formula in classical (unsorted) $FOL^=$ that depends on the benchmark's formula and background theory, and say that a benchmark is satisfiable modulo its background theory if the associated unsorted formula has a model in the classical sense. We will show later that, for background theories specified by a set of sentences, the two alternative semantics are equivalent.

6.1   Model-theoretic Semantics

[to do: definition of many-sorted structure, valuation, satisfiability in a structure, model, etc.]

Let L be an SMT-LIB logic and T its associated background theory. If Σ is the signature of T, let φ be a closed formula in L's language having signature $\Omega \supseteq \Sigma$. We say that φ is *satisfiable in L* or, equivalently that it is *satisfiable in T* iff φ is satisfiable in some Ω-model of T.

Note that for any set A of axioms for T, the definition above is equivalent to say that φ is satisfiable in T iff the set of closed formulas $A \cup \{\varphi\}$ has a model.
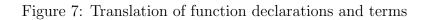
6.2   Translation Semantics

Function symbol declarations

$$
\begin{aligned}
(c : s)^\tau &= s(c) \quad \text{if } c \in \mathcal{F} \cup \mathcal{N} \cup \mathcal{R} \\
(f : s_1 \cdots s_{k+1} s)^\tau &= \forall x_1, \ldots, x_{k+1}.\, (s_1(x_1) \wedge \cdots \wedge s_k(x_{k+1})s \Rightarrow s(f(x_1, \ldots, x_{k+1})))
\end{aligned}
$$

ite-free Terms

$$
\begin{aligned}
x^\tau &= x \\
n^\tau &= n \\
r^\tau &= r \\
(f\ t_1 \cdots t_k)^\tau &= f(t_1{}^\tau, \ldots, t_k{}^\tau) \\
(u\ \alpha_1 \cdots \alpha_k)^\tau &= u^\tau
\end{aligned}
$$

Figure 7: Translation of function declarations and terms

Formulas

$$
\begin{aligned}
\mathbf{false}^\tau &= \bot \\
\mathbf{true}^\tau &= \neg\bot \\
(\pi\ t_1 \cdots t_k)^\tau &= \pi(t_1{}^\tau, \ldots, t_k{}^\tau) \\
& \quad \text{if } \pi \in \mathcal{P} \cup \{=, \mathbf{distinct}\} \text{ and } t_1, \ldots, t_k \text{ are } \mathbf{ite}\text{-free} \\
(A[\mathbf{ite}\ \varphi\ t_1\ t_2])^\tau &= (\varphi^\tau \Rightarrow (A[t_1])^\tau) \wedge (\neg\varphi^\tau \Rightarrow (A[t_2])^\tau) \\
(\mathbf{not}\ \varphi)^\tau &= \neg\varphi_1{}^\tau \\
(\mathbf{implies}\ \varphi_1\ \varphi_2)^\tau &= \varphi_1{}^\tau \Rightarrow \varphi_k{}^\tau \\
(\mathbf{and}\ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1{}^\tau \wedge \cdots \wedge \varphi_k{}^\tau \\
(\mathbf{or}\ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1{}^\tau \vee \cdots \vee \varphi_k{}^\tau \\
(\mathbf{xor}\ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1{}^\tau \oplus \cdots \oplus \varphi_k{}^\tau \\
(\mathbf{iff}\ \varphi_1 \cdots \varphi_k)^\tau &= \varphi_1{}^\tau \Leftrightarrow \cdots \Leftrightarrow \varphi_k{}^\tau \\
(\mathbf{if}\ \varphi_0\ \mathbf{then}\ \varphi_1\ \mathbf{else}\ \varphi_2)^\tau &= (\varphi_0{}^\tau \Rightarrow \varphi_1{}^\tau) \wedge (\neg\varphi_0{}^\tau \Rightarrow \varphi_2{}^\tau) \\
(\forall\, x_1{:}s_1 \ldots x_k{:}s_k\ \varphi_0)^\tau &= \forall\, x_1, \ldots, x_k.\, (s_1(x) \wedge \cdots \wedge s_k(x_k) \Rightarrow \varphi_0{}^\tau) \\
(\exists\, x_1{:}s_1 \ldots x_k{:}s_k\ \varphi_0)^\tau &= \exists\, x_1, \ldots, x_k.\, (s_1(x) \wedge \cdots \wedge s_k(x_k) \wedge \varphi_0{}^\tau) \\
(\mathbf{let}\ x = t\ \mathbf{in}\ \varphi)^\tau &= (\varphi'\{x \mapsto t\})^\tau \\
(\mathbf{flet}\ \xi = \varphi_0\ \mathbf{in}\ \varphi)^\tau &= (\varphi'\{\xi \mapsto \varphi_0\})^\tau \\
(\psi\ \alpha_1 \cdots \alpha_k)^\tau &= \psi^\tau
\end{aligned}
$$

Figure 8: Translation of formulas

One semantics for SMT-LIB formulas and theories is provided by a translation of formulas and their signatures into formulas of $FOL^=$, classical first order logic with equality.

We define a translation operator $\_^\tau$, following a well-known relativization process for translating many-sorted logics into classical logic which turns sort symbols into predicate symbols—see for instance [?, ?]. The only difference with the classical case is that we can get overloaded symbols in unsorted translations, e.g., symbols occurring with different arities or used as both predicate and function symbols. This overloading is, however, unproblematic and can be always eliminated by a simple renaming. Formally, the SMT-LIB translation semantics is defined as follows.

Definition 6 *Let Σ be an SMT-LIB signature and let T be a theory of signature Σ axiomatized by a set Ax of (SMT-LIB) formulas. For every Ω-formula ψ with $\Omega \supseteq \Sigma$ we say that ψ is satisfiable in T iff the set*

$$\{(f : s_1 \cdots s_{k+1})^\tau \mid f : s_1 \cdots s_{k+1} \in \Omega^{\mathrm{F}}\} \cup \{\varphi^\tau \mid \varphi \in Ax\} \cup \{\psi^\tau\}$$

of $FOL^=$ formulas is satisfiable (in the classical sense).

The signature Σ and the axioms Ax in the definition above are meant to be elicited, as expected, from the specification of a benchmark containing the formula ψ, and from the specification of the benchmark's background logic and theory.

A fine point about our translation semantics is that it effectively requires the given background theory T to be defined axiomatically so that one can identify the set Ax needed in Definition 6. This however is never a problem because Ax can be always defined as the set of all closed (well-formed) formulas satisfied by every model of T.[9]

The translation operator $\_^\tau$ is defined in Figure 7 for function symbol declarations and terms, and in Figure 8 for formulas. The translation into $FOL^=$ uses a conventional syntax for $FOL^=$ sentences, with the exclusive or connective denoted by $\oplus$. On terms, the operator $\_^\tau$ is defined only for those that do not contain **ite** expressions. Occurrences of **ite**'s are eliminated from terms as shown in Figure 8. It is easy to show that $\_^\tau$ is well defined.

In Figure 8, φ' denotes a *rectified* version of φ obtained by renaming apart all quantifiers—and their bound variables—into fresh variables, while $\varphi'\{x \mapsto t\}$ denotes the formula obtained from φ' by simultaneously replacing every unbound occurrence of the variable x in φ' by the term t.[10] Similarly for $\varphi'\{\xi \mapsto \varphi_0\}$. The expression $A[\textbf{ite } \varphi \ t_1 \ t_2]$ denotes an atomic formula A containing an occurrence of the term $(\textbf{ite } \varphi \ t_1 \ t_2)$, while $A[t_i]$ $(i = 1, 2)$ denotes the formula obtained from $A[\textbf{ite } \varphi \ t_1 \ t_2]$ be replacing that occurrence of $(\textbf{ite } \varphi \ t_1 \ t_2)$ with t_i.

---

[9] For the purposes of Definition 6 it is irrelevant whether Ax is *recursively* axiomatizable or not.

[10] This is to keep the (free) variables of t from becoming bound after the substitution.

6.3 Equivalence of the two Semantics

[to do]

7 Concrete Syntax

This section defines and explains the concrete syntax of the whole STM-LIB language. The adopted syntax is attribute-based and Lisp-like. Its design was driven more by the goal of simplifying parsing than that of facilitating human readability. Preferring ease of parsing over readability is reasonable in this context because it is expected not only that SMT-LIB benchmarks will be typically read by solvers but also that, by and large, they will be produced in the first place by automated tools like verification condition generators or translators from other formats.   An alternative concrete syntax, more readable for human users, and a translation from the current concrete syntax may be defined in a later version of the format. Another translation of the format, this time into XML syntax, is also planned for inclusion in the next version.

In the BNF-style production rules that define the concrete syntax, terminal symbols are denoted by *case sensitive* sequences of characters in `typewriter font`. Syntactical categories (non-terminal symbols) are denoted by text in angular braces and *slanted font*.

For simplicity, white space symbols are not modeled in the rules. It is understood though that, as usual, any two adjacent non-terminals in a rule are to be separated by one or more white space characters. In the case of two adjacent terminals or a terminal and a non-terminal, no white spaces are allowed in between unless one of the terminals is one of these symbols: (,), [,], (, {, and }.

The syntactical category ⟨*user_value*⟩, used in the following concrete grammar specifications for the value of user-defined annotations or attributes, is left underspecified in this document. The specification of the precise format of a particular user defined value is intentionally left to the user who decides to introduce the new attribute. The only constraint imposed on a user-defined value is that it start with an open brace and end with closed brace.   The reason for the enclosing braces is that user-defined attributes are not part of the official SMT-LIB format. A system compliant with the format is only required to accept them as input, but need not support any of them. Now, user defined attribute names are easily parsed (and possibly ignored) because they have the same format as predefined attributes but are not predefined. User-defined values are easily parsed (and possibly ignored) because they are enclosed in braces.   For increased flexibility, we follow the common practice of allowing C-style escape sequences. More concretely, the open and closed brace characters can occur within a user value provided they are preceded by the backslash symbol (as in \{). A similar convention applies to the

value of text attributes, which are enclosed in double quotes. There, escaped double quotes (\") are allowed in the text.

As with the abstract syntax, the production rules of the concrete syntax define a superset of the legal expressions. The subset of legal expressions is the one that satisfies the same constraints defined for the abstract syntax.

7.1   Terms and formulas

The concrete syntax for SMT-LIB unsorted terms and formulas is given in Figure 9 and Figure 10 with BNF production rules based on the abstract syntax rules given in Figure 1. Some notable aspects of the concrete syntax are its definition of *indexed identifiers*, identifiers of the form $a[n_1 : n_2 : \cdots : n_k]$ where a is an identifier in the usual sense in programming languages (a sequence of letters, digits and a handful of other characters, starting with a letter) followed by a sequence of one or more colon-separated numerical indexes in square brackets. Indexed identifiers are used for sort, function and predicate symbols. [Explain rationale] Term variables, formula variables and attributes are constructed from non-indexed identifiers by prepending the characters ?, $ and :, respectively.

7.2   Theories

The concrete syntax for SMT-LIB theory and logic declarations is given in Figure 11 and Figure 12 with BNF production rules based on the abstract syntax rules given in Figure 4 and Figure 5. Note that theory and logic names as well can be given as indexed identifiers.

7.3   Benchmarks

The concrete syntax for SMT-LIB benchmark declarations is given in Figure 13 with BNF production rules based on the abstract syntax rules given in Figure 6.

Legal benchmarks have to satisfy the following requirements in addition to the requirements corresponding to those imposed on the abstract syntax.

- The :extrasorts, :extrafuns and :extrapreds attributes cannot occur before the :logic attribute.

- Every sort symbol occurring in a declaration of the :extrafuns or :extrapreds attribute must occur in a previous declaration of the :extrasorts attribute.

- Declarations of the :assumption or :formula attribute cannot occur before the declaration of the :logic attribute.

⟨*simple_identifier*⟩	::=	*a sequence of letters, digits, dots (.), single quotes ('), and underscores (_), starting with a letter*
⟨*user_value_content*⟩	::=	*any sequence of printable characters where every occurrence of braces ({ }) is preceded by a backslash (\ )*
⟨*user_value*⟩	::=	{⟨*user_value_content*⟩}
⟨*numeral*⟩	::=	0
	|	*a non-empty sequence of digits not starting with* 0
⟨*rational*⟩	::=	⟨*numeral*⟩.0*⟨*numeral*⟩
⟨*indexed_identifier*⟩	::=	⟨*simple_identifier*⟩[⟨*numeral*⟩(:⟨*numeral*⟩)*]
⟨*identifier*⟩	::=	⟨*simple_identifier*⟩ | ⟨*indexed_identifier*⟩
⟨*var*⟩	::=	?⟨*simple_identifier*⟩
⟨*fvar*⟩	::=	$⟨*simple_identifier*⟩
⟨*attribute*⟩	::=	:⟨*simple_identifier*⟩
⟨*arith_symb*⟩	::=	*a non-empty sequence of the characters:* =, <, >, &, @, #, +, -, *, /, %, |, ~
⟨*fun_symb*⟩	::=	⟨*identifier*⟩ | ⟨*arith_symb*⟩
⟨*pred_symb*⟩	::=	⟨*identifier*⟩ | ⟨*arith_symb*⟩ | distinct
⟨*sort_symb*⟩	::=	⟨*identifier*⟩
⟨*annotation*⟩	::=	⟨*attribute*⟩ | ⟨*attribute*⟩ ⟨*user_value*⟩
⟨*base_term*⟩	::=	⟨*var*⟩ | ⟨*numeral*⟩ | ⟨*rational*⟩ | ⟨*identifier*⟩
⟨*an_term*⟩	::=	⟨*base_term*⟩ | (⟨*base_term*⟩ ⟨*annotation*⟩$^+$)
	|	(⟨*fun_symb*⟩ ⟨*an_term*⟩$^+$ ⟨*annotation*⟩*)
	|	(ite ⟨*an_formula*⟩ ⟨*an_term*⟩ ⟨*an_term*⟩ ⟨*annotation*⟩*)

Figure 9: Concrete syntax for terms

$\langle prop\_atom \rangle$ ::=   `true` | `false` | $\langle fvar \rangle$ | $\langle identifier \rangle$

$\langle an\_atom \rangle$ ::= $\langle prop\_atom \rangle$ | ($\langle prop\_atom \rangle$ $\langle annotation \rangle^{+}$)
       | ($\langle pred\_symb \rangle$ $\langle an\_term \rangle^{+}$ $\langle annotation \rangle^{*}$)

$\langle connective \rangle$ ::=   `not` | `implies` | `if_then_else`
       | `and` | `or` | `xor` | `iff`

$\langle quant\_symb \rangle$ ::=   `exists` | `forall`

$\langle quant\_var \rangle$ ::=   ($\langle var \rangle$ $\langle sort\_symb \rangle$)

$\langle an\_formula \rangle$ ::= $\langle an\_atom \rangle$
       | ($\langle connective \rangle$ $\langle an\_formula \rangle^{+}$ $\langle annotation \rangle^{*}$)
       | ($\langle quant\_symb \rangle$ $\langle quant\_var \rangle^{+}$ $\langle an\_formula \rangle$ $\langle annotation \rangle^{*}$)
       | (`let` ($\langle var \rangle$ $\langle an\_term \rangle$) $\langle an\_formula \rangle$ $\langle annotation \rangle^{*}$)
       | (`flet` ($\langle fvar \rangle$ $\langle an\_formula \rangle$) $\langle an\_formula \rangle$ $\langle annotation \rangle^{*}$)

Figure 10: Concrete syntax for formulas

$\langle string\_content \rangle$ ::=   *any sequence of printable characters where every occurrence*
       of double quotes (") is preceded by a backslash (\ )

$\langle string \rangle$ ::=   `"`$\langle string\_content \rangle$`"`

$\langle fun\_symb\_decl \rangle$ ::=   ($\langle fun\_symb \rangle$ $\langle sort\_symb \rangle^{+}$ $\langle annotation \rangle^{*}$)

$\langle pred\_symb\_decl \rangle$ ::=   ($\langle pred\_symb \rangle$ $\langle sort\_symb \rangle^{*}$ $\langle annotation \rangle^{*}$)

$\langle theory\_name \rangle$ ::= $\langle identifier \rangle$

$\langle theory\_attribute \rangle$ ::=   `:sorts` ($\langle sort\_symb \rangle^{+}$)
       | `:funs` ($\langle fun\_symb\_decl \rangle^{+}$)
       | `:preds` ($\langle pred\_symb\_decl \rangle^{+}$)
       | `:definition` $\langle string \rangle$
       | `:axioms` ($\langle an\_formula \rangle^{+}$)
       | `:notes` $\langle string \rangle$
       | $\langle annotation \rangle$

$\langle theory \rangle$ ::=   (`theory` $\langle theory\_name \rangle$ $\langle theory\_attribute \rangle^{+}$)

Figure 11: Concrete syntax for theories

$\langle logic\_name \rangle$ ::= $\langle identifier \rangle$

$\langle logic\_attribute \rangle$:=   `:theory` $\langle theory\_name \rangle$
                 | `:language` $\langle string \rangle$
                 | `:extensions` $\langle string \rangle$
                 | `:notes` $\langle string \rangle$
                 | $\langle annotation \rangle$

$\langle logic \rangle$ ::=   (`logic` $\langle logic\_name \rangle$ $\langle logic\_attribute \rangle^+$)

Figure 12: Concrete syntax for logics

$\langle status \rangle$ ::=   `sat` | `unsat` | `unknown`

$\langle bench\_name \rangle$ ::= $\langle identifier \rangle$

$\langle bench\_attribute \rangle$ ::=   `:logic` $\langle logic\_name \rangle$
                 | `:assumption` $\langle an\_formula \rangle$
                 | `:formula` $\langle an\_formula \rangle$
                 | `:status` $\langle status \rangle$
                 | `:extrasorts` ($\langle sort\_symb \rangle^+$)
                 | `:extrafuns` ($\langle fun\_symb\_decl \rangle^+$)
                 | `:extrapreds` ($\langle pred\_symb\_decl \rangle^+$)
                 | `:notes` $\langle string \rangle$
                 | $\langle annotation \rangle$

$\langle benchmark \rangle$ ::=   (`benchmark` $\langle bench\_name \rangle$ $\langle bench\_attribute \rangle^+$)

Figure 13: Concrete syntax for benchmarks

- Every sort (resp., function, predicate) symbol occurring in a declaration of the `:assumption` or `:formula` attribute must occur in a previous declaration of the `:extrasorts` (resp., `:extrafuns`, `:extrapreds`) attribute.

The ordering constraints above are imposed to simplify parsing. In fact, alternative orderings of the attributes (for instance those in which the `:extrafuns` attribute occurs before the `:logic` or the `:extrasorts` attribute) may create forward references that then require look-ahead or multiple passes to be resolved.

7.4 Comments

Source files containing SMT-LIB expressions may contain *comments* in the sense of programming languages. In SMT-LIB, a comment is a sequence of characters that starts with the character `;` and is terminated by a new line character. The choice of `;` as the comment character is simply for consistency with the Lisp-like flavor of the concrete syntax.

8 Acknowledgments

References

[1] Holger Hoos and Thomas Stützle. SATLIB–The Satisfiability Library. Web site at: `http://www.satlib.org/`.

[2] Silvio Ranise and Cesare Tinelli. SMT-LIB–The Satisfiability Modulo Theories Library. Web site at: `http://combination.cs.uiowa.edu/smtlib/`.

[3] Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library for Automated Theorem Proving. Web site at: `http://www.cs.miami.edu/~tptp/`.

A   Abstract Syntax

Terms and formulas

(Annotations)	α	::=	$a = v$
(Terms)	u	::=	$x \mid n \mid r \mid f\,t^* \mid \textbf{ite}\ \varphi\ t_1\ t_2$
(Annot. Terms)	t	::=	$u\ \alpha^*$
(Connectives)	κ	::=	$\textbf{not} \mid \textbf{implies} \mid \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \textbf{iff}$
(Atoms)	A	::=	$\textbf{true} \mid \textbf{false} \mid \xi \mid p\,t^* \mid\ = t^* \mid \textbf{distinct}\ t^*$
(Formulas)	ψ	::=	$A \mid \kappa\,\varphi^+ \mid \exists\,(x{:}s)^+\ \varphi_0 \mid \forall\,(x{:}s)^+\ \varphi_0$
		$\mid$	$\textbf{let}\ x = t\ \textbf{in}\ \varphi_0 \mid \textbf{flet}\ \xi = \varphi_0\ \textbf{in}\ \varphi_1$
		$\mid$	$\textbf{if}\ \varphi_0\ \textbf{then}\ \varphi_1\ \textbf{else}\ \varphi_2$
(Annot. Formulas)	φ	::=	$\psi\,\alpha^*$

$x \in \mathcal{X},$ the set of term variables $\xi \in \Xi,$ the set of formula variables
$n \in \mathcal{N},$ the set of numerals $r \in \mathcal{R},$ the set of positive rationals
$f \in \mathcal{F},$ the set of function symbols $p \in \mathcal{P},$ the set of predicate symbols
$s \in \mathcal{S},$ the set of sort symbols $a \in \mathcal{A},$ the set of attributes
$v \in \mathcal{V},$ the set of attribute values

Well-sorting rules for terms

$$\frac{}{\Sigma \vdash_{\text{t}} x : s}\ \text{if } x : s \in \Sigma \qquad \frac{}{\Sigma \vdash_{\text{t}} n : s}\ \text{if } n : s \in \Sigma$$

$$\frac{}{\Sigma \vdash_{\text{t}} r : s}\ \text{if } r : s \in \Sigma$$

$$\frac{\Sigma \vdash_{\text{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\text{t}} t_k : s_k}{\Sigma \vdash_{\text{t}} f\ t_1\ \cdots\ t_k : s_{k+1}}\ \text{if } f : s_1 \cdots s_{k+1} \in \Sigma$$

$$\frac{\Sigma \vdash_{\text{f}} \varphi \quad \Sigma \vdash_{\text{t}} t_1 : s \quad \Sigma \vdash_{\text{t}} t_2 : s}{\Sigma \vdash_{\text{t}} \textbf{ite}\ \varphi\ t_1\ t_2 : s}$$

Well-sorting rules for formulas

$$\frac{}{\Sigma \vdash_{\mathrm{f}} \textbf{true}} \qquad \frac{}{\Sigma \vdash_{\mathrm{f}} \textbf{false}} \qquad \frac{}{\Sigma \vdash_{\mathrm{f}} \xi}$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t_1 : s_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_k : s_k}{\Sigma \vdash_{\mathrm{f}} p \; t_1 \; \cdots \; t_k} \quad \text{if } p : s_1 \cdots s_k \in \Sigma$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t_1 : s \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_{k+2} : s}{\Sigma \vdash_{\mathrm{f}} = t_1 \; \cdots \; t_{k+2}} \qquad \frac{\Sigma \vdash_{\mathrm{t}} t_1 : s \quad \cdots \quad \Sigma \vdash_{\mathrm{t}} t_{k+2} : s}{\Sigma \vdash_{\mathrm{f}} \textbf{distinct } t_1 \; \cdots \; t_{k+2}}$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \textbf{not } \varphi} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \Sigma \vdash_{\mathrm{f}} \varphi_2}{\Sigma \vdash_{\mathrm{f}} \textbf{impl } \varphi_1 \; \varphi_2} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_0 \quad \Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \Sigma \vdash_{\mathrm{f}} \varphi_2}{\Sigma \vdash_{\mathrm{f}} \textbf{if } \varphi_0 \textbf{ then } \varphi_1 \textbf{ else } \varphi_2}$$

$$\frac{\Sigma \vdash_{\mathrm{f}} \varphi_1 \quad \cdots \quad \Sigma \vdash_{\mathrm{f}} \varphi_{k+2}}{\Sigma \vdash_{\mathrm{f}} c \; \varphi_1 \; \cdots \; \varphi_{k+2}} \quad \text{if } c \in \{\textbf{and}, \textbf{or}, \textbf{xor}, \textbf{iff}\}$$

$$\frac{\Sigma, \, x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \exists x_1{:}s_1 \ldots x{:}s_{k+1} \; \varphi} \qquad \frac{\Sigma, \, x_1{:}s_1, \ldots, x{:}s_{k+1} \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \forall x_1{:}s_1 \ldots x{:}s_{k+1} \; \varphi}$$

$$\frac{\Sigma \vdash_{\mathrm{t}} t : s \quad \Sigma, x : s \vdash_{\mathrm{f}} \varphi}{\Sigma \vdash_{\mathrm{f}} \textbf{let } x = t \textbf{ in } \varphi} \qquad \frac{\Sigma \vdash_{\mathrm{f}} \varphi_0 \quad \Sigma \vdash_{\mathrm{f}} \varphi_1}{\Sigma \vdash_{\mathrm{f}} \textbf{flet } \xi = \varphi_0 \textbf{ in } \varphi_1}$$

Theories

(Fun. sym. declaration) D_f ::= $f \; s^+ \; \alpha$

(Pred. sym. declaration) D_p ::= $p \; s^* \; \alpha$

(Attribute declaration) P_T ::= $\textbf{sorts} = s^+$
                                           | $\textbf{funs} = (D_f)^+$ | $\textbf{preds} = (D_p)^+$
                                             | $\textbf{definition} = w$ | $\textbf{axioms} = \varphi^+$
                                             | $\textbf{notes} = w$ | α

(Theory declaration) D_T ::= $\textbf{theory } T \textbf{ begin}$
                                             $(P_T)^+$
                                             $\textbf{end}$

       $T \in \mathcal{T},$ the set of theory names $w \in \mathcal{W},$ the set of character strings

Logics

(Attribute declaration)	P_L	::=	**theory** $= T$ $\mid$ **language** $= w$
		$\mid$	**extensions** $= w$ $\mid$ **notes** $= w$ $\mid$ α

(Logic declaration)	D_L	::=	**logic** L **begin**
			$(P_L)^+$
			end

$$L \in \mathcal{L}, \quad \text{the set of logic names}$$

Benchmarks

(Formula status)	σ	::=	**sat** $\mid$ **unsat** $\mid$ **unknown**

(Attribute-value pair)	P_b	::=	**logic** $= L$ $\mid$ **formula** $= \varphi$ $\mid$ **status** $= \sigma$
		$\mid$	**assumption** $= \varphi$ $\mid$ **extrasorts** $= s^+$
		$\mid$	**extrafuns** $= (D_f)^+$ $\mid$ **extrapreds** $= (D_p)^+$
		$\mid$	**notes** $= w$ $\mid$ α

(Benchmark declaration)	D_b	::=	**benchmark** b **begin**
			$(P_b)^+$
			end

$$b \in \mathcal{B}, \quad \text{the set of benchmark names}$$

B Concrete Syntax

Reserved symbols and keywords

=, and, benchmark, distinct, exists, false, flet, forall, if_then_else, iff, implies, ite, let, logic, not, or, sat, theory, true, unknown, unsat, xor.

Predefined attributes

:assumption, :axioms, :defintion, :extensions, :formula, :funs, :extrafuns, :extrasorts, :extrapreds. :language, :logic, :notes, :preds, :sorts, :status, :theory.

Terms

$\langle simple\_identifier \rangle$	::=	*a sequence of letters, digits, dots (.), single quotes ('), and underscores (_), starting with a letter*
$\langle user\_value\_content \rangle$	::=	*any sequence of printable characters where every occurrence of braces ({ }) is preceded by a backslash (\ )*
$\langle user\_value \rangle$	::=	{ $\langle user\_value\_content \rangle$ }
$\langle numeral \rangle$	::=	0
	|	*a non-empty sequence of digits not starting with* 0
$\langle rational \rangle$	::=	$\langle numeral \rangle$ .0* $\langle numeral \rangle$
$\langle indexed\_identifier \rangle$	::=	$\langle simple\_identifier \rangle$ [$\langle numeral \rangle$ (: $\langle numeral \rangle$)*]
$\langle identifier \rangle$	::=	$\langle simple\_identifier \rangle$ | $\langle indexed\_identifier \rangle$
$\langle var \rangle$	::=	? $\langle simple\_identifier \rangle$
$\langle fvar \rangle$	::=	\$ $\langle simple\_identifier \rangle$
$\langle attribute \rangle$	::=	: $\langle simple\_identifier \rangle$
$\langle arith\_symb \rangle$	::=	*a non-empty sequence of the characters:* =, <, >, &, @, #, +, -, *, /, %, |, ~
$\langle fun\_symb \rangle$	::=	$\langle identifier \rangle$ | $\langle arith\_symb \rangle$
$\langle pred\_symb \rangle$	::=	$\langle identifier \rangle$ | $\langle arith\_symb \rangle$ | distinct
$\langle sort\_symb \rangle$	::=	$\langle identifier \rangle$
$\langle annotation \rangle$	::=	$\langle attribute \rangle$ | $\langle attribute \rangle$ $\langle user\_value \rangle$
$\langle base\_term \rangle$	::=	$\langle var \rangle$ | $\langle numeral \rangle$ | $\langle rational \rangle$ | $\langle identifier \rangle$
$\langle an\_term \rangle$	::=	$\langle base\_term \rangle$ | ($\langle base\_term \rangle$ $\langle annotation \rangle^{+}$)
	|	($\langle fun\_symb \rangle$ $\langle an\_term \rangle^{+}$ $\langle annotation \rangle^{*}$)
	|	(ite $\langle an\_formula \rangle$ $\langle an\_term \rangle$ $\langle an\_term \rangle$ $\langle annotation \rangle^{*}$)

Formulas

⟨*prop_atom*⟩	::=	`true` | `false` | ⟨*fvar*⟩ | ⟨*identifier*⟩
⟨*an_atom*⟩	::=	⟨*prop_atom*⟩ | (⟨*prop_atom*⟩ ⟨*annotation*⟩⁺)
		| (⟨*pred_symb*⟩ ⟨*an_term*⟩⁺ ⟨*annotation*⟩*)

⟨*connective*⟩	::=	`not` | `implies` | `if_then_else`
		| `and` | `or` | `xor` | `iff`
⟨*quant_symb*⟩	::=	`exists` | `forall`
⟨*quant_var*⟩	::=	(⟨*var*⟩ ⟨*sort_symb*⟩)
⟨*an_formula*⟩	::=	⟨*an_atom*⟩
		| (⟨*connective*⟩ ⟨*an_formula*⟩⁺ ⟨*annotation*⟩*)
		| (⟨*quant_symb*⟩ ⟨*quant_var*⟩⁺ ⟨*an_formula*⟩ ⟨*annotation*⟩*)
		| (`let` (⟨*var*⟩ ⟨*an_term*⟩) ⟨*an_formula*⟩ ⟨*annotation*⟩*)
		| (`flet` (⟨*fvar*⟩ ⟨*an_formula*⟩) ⟨*an_formula*⟩ ⟨*annotation*⟩*)

Theories

⟨*string_content*⟩	::=	*any sequence of printable characters where every occurrence of double quotes (") is preceded by a backslash (\ )*
⟨*string*⟩	::=	`"`⟨*string_content*⟩`"`
⟨*fun_symb_decl*⟩	::=	(⟨*fun_symb*⟩ ⟨*sort_symb*⟩⁺ ⟨*annotation*⟩*)
⟨*pred_symb_decl*⟩	::=	(⟨*pred_symb*⟩ ⟨*sort_symb*⟩* ⟨*annotation*⟩*)
⟨*theory_name*⟩	::=	⟨*identifier*⟩
⟨*theory_attribute*⟩	::=	`:sorts` (⟨*sort_symb*⟩⁺)
		| `:funs` (⟨*fun_symb_decl*⟩⁺)
		| `:preds` (⟨*pred_symb_decl*⟩⁺)
		| `:definition` ⟨*string*⟩
		| `:axioms` (⟨*an_formula*⟩⁺)
		| `:notes` ⟨*string*⟩
		| ⟨*annotation*⟩
⟨*theory*⟩	::=	(`theory` ⟨*theory_name*⟩ ⟨*theory_attribute*⟩⁺)

Logics

$$\begin{array}{lll} \langle logic\_name \rangle & ::= & \langle identifier \rangle \end{array}$$

$$\begin{array}{lll} \langle logic\_attribute \rangle & := & \texttt{:theory}\ \langle theory\_name \rangle \\ & | & \texttt{:language}\ \langle string \rangle \\ & | & \texttt{:extensions}\ \langle string \rangle \\ & | & \texttt{:notes}\ \langle string \rangle \\ & | & \langle annotation \rangle \end{array}$$

$$\begin{array}{lll} \langle logic \rangle & ::= & (\ \texttt{logic}\ \langle logic\_name \rangle\ \langle logic\_attribute \rangle^{+}\ ) \end{array}$$

Benchmarks

$$\begin{array}{lll} \langle status \rangle & ::= & \texttt{sat}\ |\ \texttt{unsat}\ |\ \texttt{unknown} \end{array}$$

$$\begin{array}{lll} \langle bench\_name \rangle & ::= & \langle identifier \rangle \end{array}$$

$$\begin{array}{lll} \langle bench\_attribute \rangle & ::= & \texttt{:logic}\ \langle logic\_name \rangle \\ & | & \texttt{:assumption}\ \langle an\_formula \rangle \\ & | & \texttt{:formula}\ \langle an\_formula \rangle \\ & | & \texttt{:status}\ \langle status \rangle \\ & | & \texttt{:extrasorts}\ (\ \langle sort\_symb \rangle^{+}\ ) \\ & | & \texttt{:extrafuns}\ (\ \langle fun\_symb\_decl \rangle^{+}\ ) \\ & | & \texttt{:extrapreds}\ (\ \langle pred\_symb\_decl \rangle^{+}\ ) \\ & | & \texttt{:notes}\ \langle string \rangle \\ & | & \langle annotation \rangle \end{array}$$

$$\begin{array}{lll} \langle benchmark \rangle & ::= & (\ \texttt{benchmark}\ \langle bench\_name \rangle\ \langle bench\_attribute \rangle^{+}\ ) \end{array}$$

Index