SMTtoTPTP – A Converter for Theorem Proving Formats

Peter Baumgartner

NICTA* and Australian National University, Canberra, Australia

Abstract. *SMTtoTPTP* is a converter from proof problems written in the SMT-LIB format into the TPTP TFF format. The SMT-LIB format supports polymorphic sorts and frequently used theories like those of uninterpreted function symbols, arrays, and certain forms of arithmetics. The TPTP TFF format is an extension of the TPTP format widely used by automated theorem provers, adding a sort system and arithmetic theories. *SMTtoTPTP* is useful for, e.g., making SMT-LIB problems available to TPTP system developers, and for making TPTP systems available to users of SMT solvers. This paper describes how the conversion works, its functionality and limitations.

1 Introduction

In the automating reasoning community two major syntax formats have emerged for specifying logical proof problems. They are part of the larger infrastructure initiatives SMT-LIB [1] and TPTP [5], respectively. Both formats are under active development and are widely used for problem libraries and in competitions; both serve as *defacto* standards in the sub-communities of SMT solving and first-order logic theorem proving, respectively.

Over the last years, the theorem provers developed in the mentioned communities have become closer in functionality. SMT solvers increasingly provide support for quantified first-order logic formulas, and first-order logic theorem provers increasingly support reasoning modulo built-in theories, such as integer or rational arithmetic. Likewise, the major respective problem libraries have grown (also) by overlapping problems, i. e., problems that could be fed into both an SMT solver and a first-order theorem prover. This convergence is also reflected in recent CASC competitions. Since 2011, CASC features a competition category comprised of typed first-order logic problems modulo arithmetics (TFA), in which both SMT solvers and first-order logic theorem provers participate.

With these considerations it makes sense to provide a converter between problem formats. In this paper I focus on the more difficult direction, from the SMT-LIB format to the appropriate TPTP format, the *typed first-order TPTP format with arithmetics*, TFF [6]. This converter, *SMTtoTPTP*, is meant to be

* NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

useful for, e. g., making the existing large SMT-LIB problem libraries available to (developers of) TPTP systems, and, perhaps more importantly, making TPTP systems available to users used to SMT-LIB. *SMTtoTPTP* may also help embed TPTP systems as sub-systems in other systems that use SMT-LIB as an interface language, e.g., interactive proof assistants. However, the *SMTtoTPTP* support for that needs to remain partial as some SMT-LIB commands, e.g. those related to proof management, are not translatable into TPTP.

On notation: the SMT-LIB documents speak about *sorts* whereas in the TPTP world one has *types*. I use both terms below in their corresponding contexts, which is the only difference for the purpose of this paper.

An *operator* is either a function or a predicate symbol. Unlike the TFF format, SMT-LIB does not formally distinguish between the boolean and other sorts. Hence, all SMT-LIB operators are function symbols.

2 SMT-LIB and TFF

This section provides brief overviews of the SMT-LIB and the TFF formats as far as is needed to make this paper self-contained. See [4,1] and [7,6], respectively, for comprehensive documentation.

SMT-LIB. SMT-LIB provides a language for writing terms and formulas in a sorted (i.e., typed) version of first-order logic, a language for specifying background theories, and a language for specifying logics, i.e., suitably restricted classes of formulas to be checked for satisfiability with respect to a specific background theory. SMT-LIB also provides a command language for interacting with SMT solvers.

All SMT-LIB languages come in a Lisp-like syntax. Assuming a fixed library of logics, an SMT-LIB user writes a *script*, i.e., a sequence of *commands* to specify one or more proof problems for a given logic. A script typically contains commands for sort declarations and definitions, as well as function symbol declarations and definitions. Furthermore it will contain commands for asserting the formulas that make up the proof problem, *assertions* for short.

Sort declarations introduce new sorts by stating their name and arity, e.g., `(Pair 2)`. Sort definitions introduce new sorts in terms of already defined/declared ones. They can be parametric in sort parameters (see Example 2.1 below). Recursive definitions are disallowed.

Function symbol declarations introduce new function symbols together with their argument and result sorts given as expressions over the declared and defined sorts. Again, recursive definitions are disallowed. The semantics of function definitions is given by expansion, i.e., by in lining the definitions everywhere in the script until only declared function symbols remain.

The semantics of (sort parametric) sort definitions is given by expansion, too. Indeed, the SMT-LIB type system is *not* polymorphic. Polymorphism is not a part of the type system, it is a meta-level concept. After expansion of definitions, annotations cannot contain sort parameters, and any (well-sorted) subterm has a sort constructed of declared sorts only.

Example 2.1. The following script demonstrates the SMT-LIB type system:

```
1  (set-logic UFLIA)
2  (declare-sort Pair 2)
3  (define-sort Int-Pair (S) (Pair Int S))
4  (declare-sort Color 0)
5  (declare-fun red () Color)
6  (declare-fun get-int ((Int-Pair Color)) Int)
7  (declare-fun int-color-pair (Int Color) (Pair Int Color))
8  (assert (forall ((i Int) (c Color))
9      (= (get-int (int-color-pair i c)) i)))
10 (check-sat)
```

Here, `Pair` is declared as a 2-ary sort and `Int-Pair` is a defined sort with sort parameter `S`. In line 4, `Color` is declared as a 0-ary sort. Lines 5–7 declare some function symbols. Notice the use of `Int-Pair` in line 6 in the expression `(Int-Pair Color)`, which expands into the sort `(Pair Int Color)`. Lines 8-9 contain an assertion, its formula is obvious. The `(check-sat)` command in line 9 instructs the SMT-LIB prover to check the assertions for satisfiability. □

Introducing overloaded function symbols *in scripts* is not supported. However, theories can declare ad-hoc polymorphic function symbols. Indeed, common SMT-LIB theories make heavy use of this. For example, equality (`=`), in the core theory, is of rank $S \times S \mapsto \texttt{Bool}$ for any sort S. The theory of arrays declares a binary sort constructor `Array` and select and store operators with ranks $\texttt{Array}(S,T) \times S \mapsto T$ and $\texttt{Array(S,T)} \times S \times T \mapsto \texttt{Array}(S,T)$, respectively, for any sorts S and T.

TFF. The TFF format provides a useful extension of the untyped TPTP first-order logic format by a simple many-sorted type scheme. Types are interpreted by non-empty, pairwise disjoint, domains.

The TFF format described in [6] is just the core TFF0 of a polymorphic typed first-order format TFF1 [2]. The paper [6] also extends TFF by predefined types and operators for integer, rational and real arithmetics, which is the target language of *SMTtoTPTP*.[1]

The TFF format supports declaring (0-ary) types and function and predicate symbols over predefined and these declared types. A TFF file typically contains such declarations along with axioms and conjecture formulas over the input signature given by the declarations. In a refutational setting, conjectures need to be negated before conjoining them with the axioms.

The predefined types are the mentioned arithmetic ones and a type of individuals. Equality and the arithmetic operators are ad-hoc polymorphic over the types. All user-defined, i.e, uninterpreted, operators are monomorphic. In the

[1] The correct short name of this language is "TFA", *TFF with arithmetics*. However, most of the features of the translation are arithmetics agnostic, and so I use "TFF".

input formulas only the variables need explicit typing, which happens in quantifications. Together with the signature information this is enough for checking well-typedness.

Example 2.2. The following is a TFF specification corresponding to Example 2.1. It was obtained by the *SMTtoTPTP* program.

```
1  %% Types:
2  tff('Pair', type, 'Pair[Int,Color]': $tType).
3  tff('Color', type, 'Color': $tType).
4
5  %% Declarations:
6  tff(get_int, type, get_int: 'Pair[Int,Color]' > $int).
7  tff(int_color_pair, type, int_color_pair:
8      ($int * 'Color') > 'Pair[Int,Color]').
9
10 %% Assertions:
11 %% (forall ((i Int) (c Color)) (= (get-int (int-color-pair i c)) i))
12 tff(formula, axiom,
13   ( ! [I:$int, C:'Color'] : (get_int(int_color_pair(I, C)) = I))).
```

A `tff`-triple consists of a name, a role, and a "formula", in this order. The roles used by *SMTtoTPTP* are either `type`, for declaring types and operators with their ranks, or `axiom`, for input formulas.

The TFF syntax reserves identifiers starting with capital letters for variables. Non-variable identifiers can always be written between pairs of `'`-quotes. The example above makes heavy use of that. Lines 2 and 3 declare the types `'Pair[Int,Color]'` and `'Color'`, corresponding to the sorts `(Pair Int Color)` and `Color` in Example 2.1. Notice there is no "sort" `Int-Pair`, as all occurrences of `Int-Pair`-expressions have been removed by expansion. Lines 6 and 7 declare the same function symbols as in Example 2.1, however with the sorts expanded. Finally, the asserted formula in Example 2.1 has its counterpart in TFF-syntax in lines 11-13 above. □

3 *SMTtoTPTP* Algorithm

A regular run of *SMTtoTPTP* has four stages.

Parsing. In the first stage the commands in the input file are parsed into abstract syntax trees (ASTs), one per command. The parser has been conveniently implemented with the Scala Standard Parser Combinator Library. The ASTs for declarations, definitions and assertions are built over Scala classes (rather: instances thereof) corresponding to syntactical SMT-LIB entities such as arithmetic domain elements, constants and functional terms, let-terms, ite-terms, quantifications, sort expressions, etc.

If the set logic includes the theory of arrays, or the user explicitly asks for it, the following declarations are added to the ASTs:

```
1 (declare-sort Array 2)
2 (declare-parametric-fun (I E) select ((Array I E) I) E)
3 (declare-parametric-fun (I E) store ((Array I E) I E) (Array I E))
```

The `declare-parametric-fun` command declares parametric function symbols in the obvious way. It is meant to be useful in context with other theories as well. See Section 5 below for an example. In particular it provides type-checking for parametric operators for free. The `declare-parametric-fun` command is not in the SMT-LIB. This is not a problem, however, because it is hidden from the user.

Semantic Analysis. In the second stage the ASTs are analyzed semantically. This requires decomposing the commands into their constituents, which can be programmed in a convenient way thanks to Scala's pattern matching facilities. The main result of the analysis are various tables holding signature and other information about declared and defined function symbols and sorts. With these tables, the sort of any subterm in an assertion can be computed by expansion, as explained in Section 2. This is important for type checking and in the subsequent stages.

Transformations. In the third stage several transformations on the assertions are carried out, all on the AST level.

Removal of defined functions. All function definitions are transformed into additional assertions. This is done with a universally quantified equation between the function symbol applied to the specified variables and its body. *SMTtoTPTP* thus does not expand function definitions. The rationale is to gain flexibility by letting a theorem prover later decide whether to expand or not.[2]

Let-terms. Both the SMT-LIB and TFF formats feature "let" expressions. Unfortunately they are incompatible. An SMT-LIB let expression works much like a binder for local variables as in functional programming languages. The TFF let construct is used to locally define function or predicate symbols as syntactic macros. *SMTtoTPTP* deals with this problem by transforming SMT-LIB let expressions into existentially quantified formulas over the bound variables. This requires lifting these bindings from the term level to the formula level, thereby avoiding unintended name capturing. More precisely, the transformation works as follows.

Let ϕ[`(let ((`x t`)) `s`)`] be an SMT-LIB `Bool`-sorted term, where t is the term bound to the variable x in s (x must not be free in t). For the purpose of the transformation such a term ϕ must always exist, as let-terms occur in assertions only, which are always `Bool`-sorted. Assume that ϕ is the smallest `Bool`-sorted subformula in an assertion containing a let term, written as above, and that the let-term is an outermost one. Let $\sigma(t)$ denote the sort of term t.

[2] Defined functions could also be removed by translation into TFF let-terms, but this is clumsy as it may lead to individual let-terms in every axiom and conjecture. Moreover, let-terms are not supported by many TFF systems.

If $\sigma(t)$ is not `Bool` then the let-term is removed by existential quantification. More precisely, using SMT-LIB syntax, ϕ is replaced by the formula `(exists ((`$x\rho$ $\sigma(t)$`)) (and (=` $x\rho$ t`) (`$\phi[s\rho]$`))))`, where ρ is a renaming substitution that maps x to a fresh variable. The renaming is needed to avoid unintended variable capturing when lifting x outwards, as usual.

If $\sigma(t)$ is `Bool` then the above transformation is not possible, as TFF does not support quantification over boolean variables. In this case *SMTtoTPTP* removes the let-term by substituting x by t in s.

The above step is repeated until all let-terms are removed. The actual implementation is more efficient and requires one subterm traversal per assertion only.

Alternatively to existential quantification, all let-terms could be handled by substitution. *SMTtoTPTP* does not do that, however, because it may lead to exponentially larger terms.

Ite-terms. Both the SMT-LIB and TFF formats feature "if-then-else" constructs (ite). Fortunately, they are compatible. *SMTtoTPTP* offers the option to either keep ite-expressions in place or to transform them away. The latter is useful because not all TFF systems support ite. For example, the expression `(< (+ (ite (< 1 2) 3 4) 5) 6)` is transformed into `(and (=> (< 1 2) (< (+ 3 5) 6)) (=> (not (< 1 2)) (< (+ 4 5) 6)))`.

Array axioms. The TFF format has no predefined semantics for arrays. Hence, array axioms need to be generated as needed. This is done by sort-instantiating array axiom templates, for each array-sorted term occurring in the assertions.

Example 3.1. Assume an SMT-LIB specification

```
(set-logic AUFLIA)
(declare-sort Color 0)
(declare-fun red () Color)
(declare-fun a () (Array Int Color))
(declare-fun b () (Array Int Int))
(assert (= (select a 0) red))
```

The sole array-sorted term in assertions here is `a`, which has the sort `(Array Int Color)`. The following axioms are added:

```
(forall ((a (Array Int Color)) (i Int) (e Color))
    (= (select (store a i e) i) e))
(forall ((a (Array Int Color)) (i Int) (j Int) (e Color))
    (=> (distinct i j) (= (select (store a i e) j) (select a j))))
(forall ((a (Array Int Color)) (b (Array Int Color)))
    (=> (forall ((i Int)) (= (select a i) (select b i))) (= a b)))
```

These are standard axioms for arrays with extensional equality, sorted as required. □

TFF Generation. In the fourth stage the TFF output is generated. It starts with TFF type declarations `tff(`$name^{\texttt{TFF}}$`, type,` $\sigma^{\texttt{TFF}}$`: $tType)` for every sort σ of every subterm in every assertion. The identifier σ^{TFF} is a TFF identifier for the SMT-LIB sort σ. The TFF type identifier σ^{TFF} is merely a print representation of the sort σ, and $name^{\text{TFF}}$ is a prefix of that. As special cases, the predefined arithmetic SMT-LIB sorts `Int` and `Real` are taken as the TFF types `$int` and `$real`, respectively.

Next, a TFF type declaration consisting of the name and rank is emitted for every operator occurring in the assertions. As explained in Section 2, SMT-LIB theories may declare polymorphic function symbols. The equality and arithmetic function symbols pose no problems as these have direct counterparts in TFF. Array expressions, however, involving the polymorphic select and store operators need monomorphization and axioms for each monomorphized operator.

Monomorphization is done by including the operators rank in the name of the TFF operator. More precisely, if `(`$f\ t_1\ \cdots\ t_n$`)` is an application of the polymorphic operator f to terms $t_1, \ldots, t_n$, then *SMTtoTPTP* synthesizes an identifier, conveniently a valid TFF one, $f^{\text{TFF}} =$ `'`f`:(`$\sigma(t_1)$`*`$\cdots$`*`$\sigma(t_n)$`>`σ_{n+1}`)'` . The sort σ_{n+1} is the result sort of the term `(`$f\ t_1\ \cdots\ t_n$`)` which is obtained by applying the declaration of f to $\sigma(t_1), \ldots, \sigma(t_n)$. The rank of the operator f^{TFF} to be declared in the generated TFF hence is $\sigma_1^{\text{TFF}} \times \cdots \times^{\text{TFF}} \mapsto \sigma_{n+1}^{\text{TFF}}$. Notice the identifier f^{TFF} contains enough information to distinguish it from other applications of f with different sorts.

A special case occurs when the result sort in the declaration of f contains a free sort parameter. To avoid an error, explicit coercion is needed. For example, the "empty list of integers" could correctly be expressed as the term `(as empty (List Int))`, cf. Section 5. Monomorphization respects such coercions.

Finally, each assertion is written out as a TFF axiom. The axioms are obtained by recursively traversing the assertions' subterms and converting them into TFF terms and formulas. By and large this is straightforward translation from one syntax into another. Some comments:

- The SMT-LIB and TFF syntax of, e.g., operators and variables are rather different. *SMTtoTPTP* tries to re-use the given SMT-LIB identifiers without or only little modifications in the generated TFF. For example, an SMT-LIB variable can often be turned into a TFF variable by capitalizing the first letter.
- Certain SMT-LIB operators are varyadic and carry attributes like `chainable`, `associative` or `pairwise`. These attributes say how to translate n-ary terms over these operators into binary ones. For example, the equality operator is `chainable`: an expression `(=` $t_1\ \cdots\ t_n$`)` is first expanded into the conjunction `(and (=` $t_1\ t_2$`)` $\cdots$ `(=` $t_{n-1}\ t_n$`)` before converted to TFF.

  An exception is the `distinct` operator, which has a `pairwise` attribute. An expression `(distinct` $t_1\ \cdots\ t_n$`)` is optionally directly translated into the TFF counterpart using the `$distinct` predicate symbol. However, `$distinct` can be used only as a fact, not under any connective. If not at the top-level

of an assertion, the expression is translated into the conjunction of the expressions `(not (= ` t_i t_j `))`, for all $i, j = 1, \ldots, n$ with $i \neq j$.
- The array operators select and store are monomorphized.
- SMT-LIB equations between `Bool`-sorted terms are turned into bi-implications.
- `set-option` commands carry over their argument into a TFF comment. For example, `(set-option :answer 42)` translates into `%$ :answer 42`. Some options control the behaviour of *SMTtoTPTP*, e.g., whether to expand ite-terms or not.

Example 3.2. Example 3.1 above is converted into the following TFF. The last two array axioms are omitted for space reasons.

```
1  %% Types:
2  tff('Color', type, 'Color': $tType).
3  tff('Array', type, 'Array[Int,Color]': $tType).
4
5  %% Declarations:
6  tff(red, type, red: 'Color').
7  tff(a, type, a: 'Array[Int,Color]').
8  tff(select, type, 'select:(Array[Int,Color]*Int)>Color':
9      ('Array[Int,Color]' * $int) > 'Color').
10 tff(store, type, 'store:(Array[Int,Color]*Int*Color)>Array[Int,Color]':
11     ('Array[Int,Color]' * $int * 'Color') > 'Array[Int,Color]').
12
13 %% Assertions:
14 %% (= (select a 0) red)
15 tff(formula, axiom,
16   ('select:(Array[Int,Color]*Int)>Color'(a, 0) = red)).
17 %% (forall ((a (Array Int Color)) (i Int) (e Color))
18 %%     (= (select (store a i e) i) e))
19 tff(formula, axiom,
20   ( ! [A:'Array[Int,Color]', I:$int, E:'Color'] :
21       ('select:(Array[Int,Color]*Int)>Color'(
22           'store:(Array[Int,Color]*Int*Color)>Array[Int,Color]'(
23               A, I, E), I) = E))).
```

□

4 Limitations

SMTtoTPTP is meant to support a comprehensive subset of the SMT-LIB language and the logics and theories in [4]. Table 1 lists the SMT-LIB language elements for scripts and their status wrt. *SMTtoTPTP*.

Some of the unsupported language elements in Table 1 will be added later, e.g., indexed identifiers such as `(_ a 5)`. Other elements are intrinsicly problematic, in particular the `push` and `pop` commands. These commands are used for managing a stack of asserted formulas (typically) for incremental satisfiability

checks. This is not supported by the TPTP language, and hence *SMTtoTPTP* throws an error on encountering a `push` or `pop` command. All other commands (e.g., `get-proof`) are untranslatable and can be ignored.

Logics	
Supported: `[QF_][A][UF][(L|N)(IA|RA|IRA)]`	Unsupported: bitvectors, difference logic
Commands	
Supported: `set-logic`, `declare-sort`, `define-sort`, `declare-fun`, `define-fun`, `assert`, `exit`	Unsupported: `push`, `pop` All other commands ignored
Tokens	
Supported: *numeral*, *decimal*, *symbol*	Unsupported: *hexadecimal*, *binary*, *string*
Other Elements	
Unsupported: indexed identifiers, logic declarations, theory declarations	

Table 1. Supported SMT-LIB script language elements.

SMTtoTPTP supports a fixed set of logics. The regular expression in Table 1 denotes their SMT-LIB names. For example, `QF_AUFLIRA` means "quantifier-free logic over the combined theories of arrays, uninterpreted function symbols, and mixed linear and real arithmetics". Notice that every logic includes the *core* theory, which offers a comprehensive set of boolean-sorted operators, including equality and if-then-else.

SMTtoTPTP does not deal with SMT-LIB theory and logic declarations. As their semantics is described informally, *SMTtoTPTP* can not make much use of them. However, as said, the core theory and the theories of arrays, integer and real arithmetic are built-in.

5 Extensions

Inspired by the Z3 SMT solver [3], *SMTtoTPTP* extends the SMT-LIB standard by datatype definitions. Datatypes can be used to define enumeration types, tuples, records, and recursive data structures like lists, to name a few. The syntax of datatype definitions involves sort parameters and the constructors and destructors for elements of the datatype. Here are some examples:

```
1 (declare-datatypes () ((Color red green blue)))
2 (declare-datatypes (S T) ((Pair (mk-pair (first S) (second T)))))
3 (declare-datatypes (T) ((List nil (insert (head T) (tail (List T))))))
```

Line 1 defines an enumeration datatype with three constructors, as stated. Line 2 defines pairs over the product type `S`×`T`, where `S` and `T` are type parameters. Line 3 defines the usual polymorphic list datatype, where `nil` and `insert` are constructors, and `head` and `tail` are the destructors for the `insert`-case.

The conversion to TFF of the list datatype with a `(List Int)` sort instance, for example, is equivalent to the conversion of the following SMT-LIB commands:

```
(declare-sort List 1)
(declare-parametric-fun (T) nil () (List T))
(declare-parametric-fun (T) insert (T (List T)) (List T))
(declare-parametric-fun (T) head ((List T)) T)
(declare-parametric-fun (T) tail ((List T)) (List T))
(assert (forall ((L (List Int)))
    (or (= L (as nil (List Int))) (= L (insert (head L) (tail L))))))
(assert (forall ((N Int) (L (List Int))) (= (head (insert N L)) N)))
(assert (forall ((N Int) (L (List Int))) (= (tail (insert N L)) L)))
(assert (forall ((N Int) (L (List Int)))
    (not (= (as nil (List Int)) (insert N L)))))
```

SMTtoTPTP does not do type inference. All occurrences of type-ambiguous constructor terms must be explicitly cast to the proper sort. In the list example, (only) `nil` terms must be explicitly cast, as in `(as nil (List Int))`.

The theory of arrays has been extended with constant arrays, i.e., arrays that have the same element everywhere.

6 Other Features

SMTtoTPTP is available at `https://bitbucket.org/peba123/smttotptp` under a GNU General Public License. The distribution includes the Scala[3] source code and a ready-to-run Java jar-file. *SMTtoTPTP* can also be used as a library for parsing SMT-LIB files into an abstract syntax tree.

References

1. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, eds., *SMT Workshop*, 2010.
2. J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form With Rank-1 Polymorphism. In M. P. Bonacina, ed., *CADE-24*, *LNCS 7898*, pp 414–420. Springer, 2013.
3. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, eds., *TACAS*, *LNCS 4963*, pp 337–340. Springer, 2008.
4. SMT-LIB, The Satisfiability Modulo Theories Library. `http://smt-lib.org/`.
5. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
6. G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In N. Bjørner and A. Voronkov, eds., *LPAR-18*, *LNCS 7180*. Springer, 2012.
7. The TPTP Problem Library for Automated Theorem Proving. `http://www.cs.miami.edu/~tptp/`.

[3] `http://www.scala-lang.org`