The TPTP Extended Typed First-order Form - TFX

Geoff Sutcliffe[1] and Evgenii Kotelnikov[2]

[1] University of Miami, USA
[2] Chalmers University of Technology, Sweden

Abstract

1 Introduction

The TPTP world [11] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP world includes the TPTP problem library, the TSTP solution library, the TMTP model library, standards for writing ATP problems and reporting ATP solutions, tools and services for processing ATP problems and solutions, and it supports the CADE ATP System Competition (CASC). Various parts of the TPTP world have been deployed in a range of applications, in both academia and industry. The web page `http://www.tptp.org` provides access to all components.

The TPTP language is one of the keys to the success of the TPTP world. The language is used for writing both TPTP problems and TSTP solutions, which enables convenient communication between different systems and researchers. Originally the TPTP world supported only first-order clause normal form (CNF) [14]. Over the years support for full first-order form (FOF) [10], monomorphic typed first-order form (TF0) [13], rank-1 polymorphic typed first-order form (TF1) [3], monomorphic typed higher-order form (TH0) [12], and rank-1 polymorphic typed higher-order form (TH1) [5], have been added. (TF0 and TF1 together form the TFF language family; TH0 and TH1 together form the THF language family.)

Since the inception of TFF and THF there have been some features that have received little use, and hence little attention. In particular, tuples, conditional expressions (if-then-else), and let expressions (let-defn-in) were neglected, and in TFF they were horribly formulated with variants to distinguish between their use with formulae and terms. Recently, conditional expressions and let expressions have become more important because of their use in software verification applications. In an independent development, Evgenii Kotelnikov et al. introduced the FOOL logic [8], which extends first-order logic so that (i) formulae can be used as terms of the boolean type, (ii) variables of the boolean type can be used as formulae, (iii) tuple terms and tuple types are available as first-class citizens, and (iv) conditional and let expressions are supported. This logic can be automatically translated to ordinary many-sorted first-order logic [8]. Features of FOOL can be used to concisely express problems coming from program analysis [9] or translated from more expressive logics. Conditional expressions and let expressions of FOOL resemble those of the SMT-LIB language version 2 [1].

The TPTP's new Typed First-order form eXtended (TFX) language remedies the old weaknesses of TFF, and incorporates the features of FOOL logic. This has been achieved by conflating (with some exceptions) formulae and terms, simplifying tuples in plain TFF, including fully expressive tuples in TFX, removing the old conditional expressions and let expressions from TFF, and including new elegant forms of conditional expressions and let expressions as part of TFX (this more elegant form has been mirrored in THF, as described below). TFX is a superset of the TFF language.

This paper describes the extensions to the TFF language form that define the TFX language, and changes to the THF language that correspond to decisions made for TFX. The remainder of this paper is organized as follows: Section 2 reviews the TFF and THF languages, and describes the FOOL logic. Section 3 describes the new features of TFX and their syntax, and the syntax changes made to the THF language. Section 4 describes the evolving software support for TFX, and provides some examples that illustrate it's use. Section 5 concludes.

2 The TPTP Language and the FOOL Logic

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language, suitable for writing both ATP problems and solutions. The top level building blocks of the TPTP language are *annotated formulae*. An annotated formula has the form:

language`(`*name*`,` *role*`,` *formula*`,` `[`*source*`,` `[`*useful_info*`]])`.

The *language*s supported are clause normal form (`cnf`), first-order form (`fof`), typed first-order form (`tff`), and typed higher-order form (`thf`). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula in an ATP system. In the *formula*, terms and atoms follow Prolog conventions, i.e., functions and predicates start with a lowercase letter or are `'single quoted'`, variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a `$`, or are composed of non-alphanumeric characters, e.g., the truth constants `$true` and `$false`, and integer/rational/real numbers such as 27, 43/92, -99.66. The basic logical connectives are `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for $\forall$, $\exists$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Equality and inequality are expressed as the infix operators `=` and `!=`. An example annotated first-order formula, supplied from a file, is:

```
fof(union,axiom,
  ( ! [X,A,B] :
      ( member(X,union(A,B))
    <=> ( member(X,A)
        | member(X,B) ) )
  file('SET006+0.ax',union),
  [description('Definition of union'), relevance(0.9)]).
```

2.1 The Typed First-order Form TFF

TFF extends the basic FOF language with *types* and *type declarations*. The TF0 variant is monomorphic, and the TF1 variant is rank-1 polymorphic. Every function and predicate symbol is declared before its use, with a *type signature* that specifies the types of the symbol's arguments and result. TF0 types τ have the following forms:

- the predefined types `$i` for ι (individuals) and `$o` for o (booleans);
- the predefined arithmetic types `$int` (integers), `$rat` (rationals), and `$real` (reals);
- user-defined types (constants).

User-defined types are declared (before their use) to be of the kind `$tType`, in annotated formulae with a `type` role – see Figure 1 for examples. TF0 type signatures ς have the following forms:

- individual types τ;
- $(\tau_1$ `*` $\cdots$ `*` $\tau_n)$ `>` $\tilde{\tau}$ for $n > 0$, where τ_i are the argument types, and $\tilde{\tau}$ is the result type.

The type signatures of uninterpreted symbols are declared like types, in annotated formulae with a `type` role – see Figure 1 for examples. The type of `=` is ad hoc polymorphic over all types except `$o` (this restriction is lifted in TFX), with both arguments having the same type and the result type being `$o`. The types of arithmetic predicates and functions are ad hoc polymorphic over the arithmetic types; see [13] for details. Figure 1 illustrates some TF0 formulae, whose conjecture can be proved from the axioms (it is the TPTP problem `PUZ130_1.p`).

```
%------------------------------------------------------------------------------
tff(animal_type,type,( animal: $tType )).
tff(cat_type,type,( cat: $tType )).
tff(dog_type,type,( dog: $tType )).
tff(human_type,type,( human: $tType )).
tff(cat_to_animal_type,type,( cat_to_animal: cat > animal )).
tff(dog_to_animal_type,type,( dog_to_animal: dog > animal )).
tff(garfield_type,type,( garfield: cat )).
tff(odie_type,type,( odie: dog )).
tff(jon_type,type,( jon: human )).
tff(owner_of_type,type,( owner_of: animal > human )).
tff(chased_type,type,( chased: ( dog * cat ) > $o )).
tff(hates_type,type,( hates: ( human * human ) > $o )).

tff(human_owner,axiom,(
    ! [A: animal] :
    ? [H: human] : H = owner_of(A) )).

tff(jon_owns_garfield,axiom,(
    jon = owner_of(cat_to_animal(garfield)) )).

tff(jon_owns_odie,axiom,(
    jon = owner_of(dog_to_animal(odie)) )).

tff(jon_owns_only,axiom,(
    ! [A: animal] :
      ( jon = owner_of(A)
     => ( A = cat_to_animal(garfield) | A = dog_to_animal(odie) ) ) )).

tff(dog_chase_cat,axiom,(
    ! [C: cat,D: dog] :
      ( chased(D,C)
     => hates(owner_of(cat_to_animal(C)),owner_of(dog_to_animal(D))) ) )).

tff(odie_chased_garfield,axiom,(
    chased(odie,garfield) )).

tff(jon_hates_jon,conjecture,(
    hates(jon,jon) )).
%------------------------------------------------------------------------------
```

Figure 1: TF0 Formulae

The polymorphic TF1 extends TF0 with (user-defined) *type constructors*, *type variables*, polymorphic symbols, and one new binder. TF1 types τ have the following forms:

- the predefined types `$i` and `$o`;
- the predefined arithmetic types `$int`, `$rat`, and `$real`;
- user-defined n-ary type constructors applied to n type arguments;
- type variables, which must be quantified by `!>` – see the type signature forms below.

Type constructors are declared (before their use) to be of the kind `($tType *` $\cdots$ `* $tType) > $tType`, in annotated formulae with a `type` role. TF1 type signatures ς have the following forms:

- individual types τ;
- `(`τ_1 `*` $\cdots$ `*` τ_n`) >` $\tilde{\tau}$ for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type (with the same caveats as for TF0);
- `!>[`α_1`: $tType, ... ,`α_n`: $tType]:` ς for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς is a type signature.

The binder `!>` in the last form denotes universal quantification in the style of $\lambda\Pi$ calculi. It is only used at the top level in polymorphic type signatures. All type variables must be of type `$tType`; more complex type variables, e.g., `$tType > $tType` are beyond rank-1 polymorphism. As in TF0, arithmetic symbols and equality are ad hoc polymorphic. An example of TF1 formulae can been found in [5].

2.2 The Typed Higher-order Form THF

THF extends FOF with higher-order notions, including adoption of curried form for type declarations, lambda terms (with a lambda binder), symbol application, and new binders. The TH0 variant is monomorphic, and the TH1 variant is rank-1 polymorphic. TH0 types τ have the following forms:

- the predefined types `$i` and `$o`;
- the predefined arithmetic types `$int`, `$rat`, and `$real`;
- user-defined types (constants);
- `(`τ_1 `>` $\cdots$ `>` τ_n `>` $\tilde{\tau}$`)` for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type.

TH0 type signatures ς have the following forms:

- individual types τ.

The `(`τ_1 `>` $\cdots$ `>` τ_n `>` $\tilde{\tau}$`)` form is promoted to be a type in TH0 (and TH1) forms, so that variables in formulae can be quantified as function types. The curried form means that TH0 provides the possibility of partial application. As in TF0, arithmetic symbols and equality are ad hoc polymorphic. The new binary connective `@` represents application. There are three new binders: `^`, `@+`, and `@-`, for λ, ϵ (indefinite description, aka choice), and ι (definite description). Use of all the connectives as terms is also supported. The semantics for TH0 is Henkin semantics with choice (Henkin semantics by definition also includes Boolean and functional extensionality) [2, 4]. Figure 2 illustrates some TH0 formulae, whose conjecture can be proved from the axioms (it is a variant of the TPTP problem `PUZ140^1.p`).

The polymorphic TH1 combines the higher-order features of TH0 with the polymorphic features of TF1. TH1 also adds five new polymorphic constants. TH1 types τ have the following forms:

- the predefined types `$i` and `$o`;

```
%------------------------------------------------------------------------------
thf(syrup_type,type,( syrup: $tType )).
thf(beverage_type,type,( beverage: $tType )).
thf(coffee_type,type,( coffee: beverage )).
thf(mix_type,type,( mix: beverage > syrup > beverage )).
thf(coffee_mixture_type,type,( coffee_mixture: syrup > beverage )).
thf(hot_type,type,( hot: beverage > $o )).

%----The mixture of coffee and something
thf(coffee_mixture_definition,definition,
    ( coffee_mixture = ( mix @ coffee ) )).

%----Any coffee mixture is hot coffee
thf(coffee_and_syrup_is_hot_coffee,axiom,(
    ! [S: syrup] : ( ( (coffee_mixture @ S) = coffee )
                   & ( hot @ ( coffee_mixture @ S ) )) )).

%----There is some mixture of coffee and any syrup which is hot coffee
thf(there_is_hot_coffee,conjecture,(
    ? [SyrupMixer: syrup > beverage] :
    ! [S: syrup] :
    ? [B: beverage] :
      ( ( B = ( SyrupMixer @ S ) ) & ( B = coffee ) & ( hot @ B ) ) )).
%------------------------------------------------------------------------------
```

Figure 2: TH0 Formulae

- user-defined n-ary type constructors applied to n type arguments;
- type variables (which must be quantified by `!>`);
- $(\tau_1$ `>` $\cdots$ `>` τ_n `>` $\tilde{\tau})$ for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type.

TH1 type signatures ς have the following forms:

- individual types;
- `!>[`α_1`: $tType, ... ,`α_n`: $tType]:` ς for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς is a type signature.

TH1 has five new polymorphic constants: `!!` for Π (universal quantification), `??` for Σ (existential quantification), `@@+` for ϵ (indefinite description, aka choice), `@@-` for ι (definite description), and `@=` (equality). Each of these must be instantiated by applying them to exactly one type argument. An example of TH1 formulae can been found in [5].

2.3 The FOOL Logic

FOOL [8], standing for First-Order Logic (FOL) + bOoleans, is a variation of many-sorted first-order logic in which (i) formulae can be used as terms of the boolean type, (ii) variables of the boolean type can be used as formulae, (iii) tuple terms and tuple types are available as first-class citizens, and (iv) conditional and let expressions are supported. There is a model-preserving transformation of FOOL formulae to FOL formulae [8], so that an implementation of the transformation makes it possible to prove FOOL formulae using a regular first-order theorem prover. Formulae of FOOL can also be efficiently translated to a first-order clausal normal form [7].

In the following subsections we summarise the features of FOOL and illustrate them using examples taken from [6] and [9].

2.3.1 First-class boolean sort

FOOL contains a fixed two-element type *bool*, allows quantification over variables of type *bool*, and considers terms of type *bool* to be formulae. Formula 1 is a syntactically correct tautology in FOOL.

$$(\forall x : \mathit{bool})(x \vee \neg x) \tag{1}$$

The existence of *bool* terms means that a function or a predicate can take a formula as an argument, and formulae can be used as arguments to equality. For example, one can define the logical implication as a binary function *impl* of the type $\mathit{bool} \times \mathit{bool} \rightarrow \mathit{bool}$ using the following axiom.

$$(\forall x : \mathit{bool})(\forall y : \mathit{bool})(\mathit{impl}(x, y) \Leftrightarrow \neg x \vee y). \tag{2}$$

Formula 2 can be equivalently expressed with $\doteq$ instead of $\Leftrightarrow$. Then it is possible to express that P is a graph of a (partial) function of the type $\sigma \rightarrow \tau$ as follows.

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau)\mathit{impl}(P(x, y) \wedge P(x, z), y \doteq z) \tag{3}$$

2.3.2 if-then-else expressions

FOOL contains expressions of the form $\texttt{if}\ \psi\ \texttt{then}\ s\ \texttt{else}\ t$, where ψ is a boolean term, and s and t are terms of the same sort. The semantics of such expressions mirrors the semantics of conditional expressions in programming languages, and are therefore convenient for expressing formulae coming from program analysis. For example, consider the *max* function of the type $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ that returns the maximum of its arguments. Its definition can be expressed in FOOL as

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(\mathit{max}(x, y) \doteq \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y). \tag{4}$$

FOOL allows if-then-else expressions to occur as formulae, as in the following valid property of *max*

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(\texttt{if}\ \mathit{max}(x, y) \doteq x\ \texttt{then}\ x \geq y\ \texttt{else}\ y \geq x). \tag{5}$$

2.3.3 let-in expressions

FOOL contains expressions of the form $\texttt{let}\ D_1; \ldots; D_k\ \texttt{in}\ t$, where $k > 0$, t is either a term or a formula, and $D_1, \ldots, D_k$ are simultaneous non-recursive definitions. FOOL allows definitions of function symbols, predicate symbols, and tuples.

A definition of a function symbol $f : \sigma_1 \times \ldots \times \sigma_n \rightarrow \tau$ has the form $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s$, where $n \geq 0$, and s is a term of the sort τ. The following let-in expression denotes the maximum of three integer constants a, b, and c using a local definition of the function symbol *max*.

$$\begin{array}{l}\texttt{let}\ \mathit{max}(x : \mathbb{Z}, y : \mathbb{Z}) = \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y \\ \ \texttt{in}\ \mathit{max}(\mathit{max}(a, b), c)\end{array} \tag{6}$$

A definition of a predicate symbol $p : \sigma_1 \times \ldots \times \sigma_n$ has the form $p(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = \phi$, where $n \geq 0$, and ϕ is a formula. The following let-in expression denotes equivalence of two boolean constants A and B using a local definition of the predicate symbol *impl*.

$$\begin{aligned}&\texttt{let}\ \mathit{impl}(x : \mathit{bool}, y : \mathit{bool}) = \neg x \vee y\\ &\ \texttt{in}\ \mathit{impl}(A, B) \wedge \mathit{impl}(B, A)\end{aligned} \tag{7}$$

A definition of a tuple has the form $(c_1, \ldots, c_n) = s$, where $n > 1$, $c_1, \ldots, c_n$ are constant symbols of the sorts $\sigma_1, \ldots, \sigma_n$, respectively, and s is a tuple expression. A tuple expression is inductively defined to be either

1. $(s_1, \ldots, s_n)$, where $s_1, \ldots, s_n$ are terms of the sorts $\sigma_1, \ldots, \sigma_n$, respectively;
2. $\texttt{if}\ \phi\ \texttt{then}\ s_1\ \texttt{else}\ s_2$, where ϕ is a formula, and s_1 and s_2 are tuple expressions; or
3. $\texttt{let}\ D_1; \ldots; D_k\ \texttt{in}\ s'$, where $D_1; \ldots; D_k$ are definitions, and s' is a tuple expression.

The semantics of let-in expressions in FOOL mirrors the semantics of simultaneous non-recursive local definitions in programming languages. That is, neither of the definitions $D_1, \ldots, D_n$ uses function or predicate symbols created by any other definition. In the following example, constants a and b are swapped by a let-in expression. The resulting formula is equivalent to $f(b, a)$.

$$\texttt{let}\ a = b\,;\, b = a\ \texttt{in}\ f(a, b) \tag{8}$$

The main application of let-in expressions with tuple definitions is in problems coming from program analysis, namely modelling of assignments [9]. The lefthand side of Figure 3 shows an example of an imperative `if` statement containing assignments to integer variables, and an `assert` statement. This can be encoded as a FOOL formula as shown on the righthand side, using let-in expressions with definitions of tuples that capture the assignments.

```
if (x > y)
    t := x;
    x := y;
    y := t;

assert x <= y;
```

$$\begin{aligned}\texttt{let}\ (x, y, t) = {}& \texttt{if}\ x > y\\ & \texttt{then}\ \texttt{let}\ t = x\ \texttt{in}\\ & \qquad\quad \texttt{let}\ x = y\ \texttt{in}\\ & \qquad\qquad \texttt{let}\ y = t\ \texttt{in}\\ & \qquad\qquad\quad (x, y, t)\\ & \texttt{else}\ (x, y, t)\\ \texttt{in}\ x \leq y\ & \end{aligned}$$

Figure 3: FOOL encoding of an `if` statement

3 The TFX Syntax

Tuples, numbers, and "distinct objects" are terms but not formulae. All formulae are terms. From the point of view of the first class boolean sort terms and formulae are fully conflated.

4 Implementation and Examples

5 Conclusion

This paper has described the polymorphic typed-higher order form (TH1) in the TPTP language family, which extends the monomorphic TH0 language with TF1-style rank-1 polymorphism.

Acknowledgements. Thanks to Jasmin Blanchette and Christoph Benzmüller for their constructive input, and to Jasmin Blanchette and Andrei Paskevich for letting us plagerize parts of their TF1 paper [3]. Kaliszyk was supported by the Austrian Science Fund grant P26201.

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[2] C. Benzmüller, C.E. Brown, and M. Kohlhase. Higher-order Semantics and Extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[3] J. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-order Form with Rank-1 Polymorphism. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 414–420. Springer-Verlag, 2013.

[4] L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[5] C. Kaliszyk, G. Sutcliffe, and F. Rabe. TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Proceedings of the 5th Workshop on the Practical Aspects of Automated Reasoning*, number 1635 in CEUR Workshop Proceedings, pages 41–55, 2016.

[6] E. Kotelnikov, L. Kovacs, G. Reger, and A. Voronkov. The Vampire and the FOOL. In J. Avigad and A. Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 37–48. ACM, 2016.

[7] E. Kotelnikov, L. Kovacs, M. Suda, and A. Voronkov. A Clausal Normal Form Translation for FOOL. In C. Benzmüller, G. Sutcliffe, and R. Rojas, editors, *Proceedings of the 2nd Global Conference on Artificial Intelligence*, number 41 in EPiC Series in Computing, pages 53–71, 2016.

[8] E. Kotelnikov, L. Kovacs, and A. Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Proceedings of the International Conference on Intelligent Computer Mathematics*, number 9150 in Lecture Notes in Computer Science, pages 71–86. Springer-Verlag, 2015.

[9] E. Kotelnikov, L. Kovacs, and A. Voronkov. A FOOLish Encoding of the Next State Relations of Imperative Programs. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, page Submitted, 2018.

[10] 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.

[11] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

[12] G. Sutcliffe and C. Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[13] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pages 406–419. Springer-Verlag, 2012.

[14] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.