TFX: The TPTP Extended Typed First-order Form

Geoff Sutcliffe[1] and Evgenii Kotelnikov[2]

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

Abstract. The TPTP world is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving systems for classical logics. The TPTP language is one of the keys to the success of the TPTP world. Originally the TPTP world supported only first-order clause normal form (CNF). Over the years support for full first-order form (FOF), monomorphic typed first-order form (TF0), rank-1 polymorphic typed first-order form (TF1), monomorphic typed higher-order form (TH0), and rank-1 polymorphic typed higher-order form (TH1), have been added. TF0 and TF1 together form the TFF language family; TH0 and TH1 together form the THF language family. This paper introduces the eXtended Typed First-order form (TFX), which extends TFF to include boolean terms, tuples, conditional expressions, and let expressions.

1 Introduction

The TPTP world [12] 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, 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) [16]. Over the years support for full first-order form (FOF) [11], monomorphic typed first-order form (TF0) [15], rank-1 polymorphic typed first-order form (TF1) [2], monomorphic typed higher-order form (TH0) [14], and rank-1 polymorphic typed higher-order form (TH1) [4], have been added. TF0 and TF1 together form the TFF language family; TH0 and TH1 together form the THF language family. See [13] for a recent review of the TPTP.

Since the inception of TFF 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 the latter two were horribly formulated with variants to distinguish between their use as 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 FOOL [7], an extension of many-sorted first-order logic. FOOL contains (i) an interpreted boolean type, which allows boolean variables to be used as formulae, and allows all formulae to be used as boolean terms, (ii) conditional expressions, and (iii) let expressions. FOOL can be straightforwardly extended with the polymorphic theory of tuples that defines first class tuple types and terms [8]. Features of FOOL can be used to concisely express problems coming from program analysis [8] or translated from more expressive logics. The conditional expressions and let expressions of FOOL resemble those of the SMT-LIB language version 2 [1].

The TPTP's new eXtended Typed First-order form (TFX) language remedies the old weaknesses of TFF, and incorporates the features of FOOL. This has been achieved by conflating (with some exceptions) formulae and terms, removing tuples from 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. (These more elegant forms have been mirrored in THF, but that is not a topic of this paper.) TFX is a superset of the revised TFF language. This paper describes the extensions to the TFF language form that define the TFX language. The remainder of this paper is organized as follows: Section 2 reviews the TFF language, and describes FOOL. Section 3 provides technical and syntax details of the new features of TFX. Section 4 describes the evolving software support for TFX, and provides some examples that illustrate its use. Section 5 concludes.

2 The TFF Language and FOOL

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 logical connectives are `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for ∀, ∃, ¬, ∨, ∧, ⇒, ⇐, ⇔, and ⊕ respectively. Equality and inequality are expressed as the

infix operators = and !=. The following is an example of an annotated first-order formula, supplied from a file.

```
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 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. Each TF0 type is one of

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

User defined types are declared before their use to be of the kind `$tType`, in annotated formulae with the `type` role - see Figure 1 for examples. Each TF0 type signature declares either

- an individual type τ; or
- a function type $(\tau_1 \texttt{ * } \cdots \texttt{ * } \tau_n) \texttt{ > } \tilde{\tau}$ 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 the `type` role - see Figure 1 for examples. The type of = and != 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 [15] for details. Figure 1 illustrates some TF0 formulae whose conjecture can be proved from the axioms (it's the TPTP problem `PUZ130_1.p`).

The polymorphic TF1 extends TF0 with (user-defined) *type constructors*, *type variables*, polymorphic symbols, and a new binder. Each TF1 type is one of

- 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; or
- type variables, which must be quantified by `!>` - see the type signature forms below.

Type constructors are declared in annotated formulae with a `type` role before their use, to be of the kind `($tType * ⋯ * $tType) > $tType`. Each TF1 type signature declares either

- an individual type τ;
- a function type $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type; or
- a polymorphic type $!>[\alpha_1 : \$tType, \ldots, \alpha_n : \$tType] : \varsigma$ for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς is a TF0 type signature.

The `!>` binder in the last form denotes universal quantification in the style of $\lambda\Pi$ calculi. It is used only at the top level in polymorphic type signatures. All type variables must be of type `$tType`; more complex type variables are beyond rank-1 polymorphism. An example of TF1 formulae can be found in [4].

```
%----------------------------------------------------------------------
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) ).
%----------------------------------------------------------------------
```

Fig. 1. TF0 Formulae

2.2 FOOL

FOOL [7], standing for First-Order Logic (FOL) + bOoleans, is an extension of many-sorted first-order logic. FOOL contains (i) an interpreted boolean type, which allows boolean variables to be used as formulae, and allows all formulae to be used as boolean terms, (ii) conditional expressions, and (iii) let expressions. FOOL can be straightforwardly extended with the polymorphic theory of tuples that defines first class tuple types and terms [8]. In what follows we consider that extension, and tuples are part of TFX. There is a model-preserving transformation of FOOL formulae to many-sorted first-order logic [7], so that an implementation of the transformation makes it possible to reason in FOOL using an existing ATP system for many-sorted first-order logic. Formulae of FOOL can also be efficiently translated to first-order clause normal form [6]. The following describes these features of FOOL, illustrating them using examples taken from [5] and [8]. The complete formal semantics of FOOL is given in [7].

Boolean Terms and Formulae
FOOL contains an interpreted two-element boolean type *bool*, allows quantification over variables of type *bool*, and considers formulae to be terms of type *bool*. This allows boolean variables to be used as formulae, and all formulae to be used as boolean terms. For example, Formula 1 is a syntactically correct tautology in FOOL.

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

Logical implication can be defined as a binary function *imply* of the type $bool \times bool \rightarrow bool$ using the axiom

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

A graph P of a (partial) function of the type $\sigma \rightarrow \tau$ can be expressed as

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau) imply(P(x, y) \wedge P(x, z), y = z) \tag{3}$$

Formula 2 can be equivalently written with $=$ instead of $\Leftrightarrow$.

Tuples
FOOL extended with the theory of tuples contains a type $(\sigma_1, \ldots, \sigma_n)$ of the n-ary tuple for all types $\sigma_1, \ldots, \sigma_n$, $n > 0$. Each type $(\sigma_1, \ldots, \sigma_n)$ is first class, that is, it can be used in the type of a function or predicate symbol, and in a quantifier. An expression $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms of types $\sigma_1, \ldots, \sigma_n$, respectively, is a tuple term of type $(\sigma_1, \ldots, \sigma_n)$. Each tuple term is first class and can be used as an argument to a function symbol, a predicate symbol, or equality.

Tuples are ubiquitous in mathematics and programming languages. For example, one can use the tuple type $(\mathbb{R}, \mathbb{R})$ as the type of complex numbers. Thus the term $(2, 3)$ represents the complex number $2+3i$. A function symbol *plus* that represents addition of complex numbers has the type $(\mathbb{R}, \mathbb{R}) \times (\mathbb{R}, \mathbb{R}) \rightarrow (\mathbb{R}, \mathbb{R})$.

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

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

FOOL allows conditional expressions to occur as formulae, as in the following valid property of *max*.

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

Let Expressions
FOOL contains let 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.

The definition of a function symbol $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ has the form $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s$, where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables, and s is a term of the type τ. For example, the following denotes the maximum of three integer constants a, b, and c, using a local definition of the function symbol *max*.

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

The definition of a predicate symbol $p : \sigma_1 \times \ldots \times \sigma_n \to \mathit{bool}$ has the form $p(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = \varphi$, where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables, and φ is a formula. For example, the following denotes equivalence of two boolean constants A and B, using a local definition of the predicate symbol *imply*.

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

The definition of a tuple has the form $(c_1, \ldots, c_n) = s$, where $n > 1$, $c_1, \ldots, c_n$ are distinct constant symbols of the types $\sigma_1, \ldots, \sigma_n$, respectively, and s is a term of the type $(\sigma_1, \ldots, \sigma_n)$. For example, the following defines addition for complex numbers using two simultaneous local definition of tuples.

$$\begin{array}{l} (\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R})) \\ \quad (\mathit{plus}(x, y) = \texttt{let}\ (a, b) = x;\ (c, d) = y\ \texttt{in}\ (a + c, b + d)) \end{array} \tag{8}$$

The semantics of let expressions in FOOL mirrors the semantics of simultaneous non-recursive local definitions in programming languages. That is, none

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 expression, and the formula is equivalent to $P(b, a)$.

$$\texttt{let}\ a = b\,;\ b = a\ \texttt{in}\ P(a, b) \tag{9}$$

Formula 9 can be equivalently expressed by the following let expression with a definition of a tuple.

$$\texttt{let}\ (a, b) = (b, a)\ \texttt{in}\ P(a, b) \tag{10}$$

Let expressions with tuple definitions are convenient for expressing problems coming from program analysis, namely modelling of assignments [8]. The left hand side of Figure 2 shows an example of an imperative `if` statement containing assignments to integer variables, and an `assert` statement. This can be encoded in FOOL as shown on the right hand side, using let expressions with definitions of tuples that capture the assignments.

```
if (x > y) {
   t := x;
   x := y;
   y := t;
}
assert x <= y;
```

```
let (x, y, t) = if x > y
                then let t = x in
                       let x = y in
                         let y = t in
                           (x, y, t)
                else (x, y, t)
 in x ≤ y
```

Fig. 2. FOOL encoding of an `if` statement

3 The TFX Syntax

The TPTP TFX syntax extends the TFF syntax to provide the features of FOOL, and at the same time some of the previous weaknesses of plain TFF have been remedied. Formulae and terms have been conflated (with some exceptions). Tuples have been removed from TFF, and fully expressive tuples included in TFX. The old conditional expressions and let expressions have been removed from TFF, and new elegant forms have been included as part of TFX. The grammar of TFX is captured in version v7.1.0.2 of the TPTP syntax, available online at http://www.tptp.org/TPTP/SyntaxBNF.html. In the subsections below, the relevant excerpts of the BNF are provided, with examples and commentary.

3.1 Boolean Terms and Formulae

Variables of type `$o` can be used as formulae, and formulae can be used as terms. The following is the relevant BNF excerpt. Formulae and terms are conflated by including logic/atomic formulae as options for terms/unitary terms. The distinction between formulae and terms is maintained for plain TFF.

```
<tff_logic_formula>    ::= <tff_unitary_formula> | <tff_unary_formula> |
                           <tff_binary_formula> | <tff_defined_infix>
<tff_unitary_formula>  ::= <tff_quantified_formula> | <tff_atomic_formula> |
                           <tfx_unitary_formula> | (<tff_logic_formula>)
<tfx_unitary_formula>  ::= <variable>
<tff_term>             ::= <tff_logic_formula> | <defined_term> | <tfx_tuple>
<tff_unitary_term>     ::= <tff_atomic_formula> | <defined_term> |
                           <tfx_tuple> | <variable> | (<tff_logic_formula>)
```

The FOOL tautology in Formula 1 can be written in TFX as

```
tff(tautology,conjecture, ! [X: $o]: (X | ~X) ).
```

The *imply* predicate in Formula 2 can be written in TFX as

```
tff(imply_type,type,imply: ($o * $o) > $o ).
tff(imply_defn,axiom,
    ! [X: $o,Y: $o]: (imply(X,Y) <=> (~X | Y)) ).
```

The definition of a graph of a function in Formula 3 can be written in TFX as

```
tff(s,type,s: $tType).
tff(t,type,t: $tType).
tff(p,type,p: (s * t) > $o ).
tff(graph,axiom,
    ! [X: s,Y: t,Z: s] : imply(p(X,Y) & p(X,Z),Y = Z) ).
```

A consequence of allowing formulae as terms is that the default typing of functions and predicates supported in plain TFF (functions default to `($i * ...* $i) > $i` and predicates default to `($i * ...* $i) > $o`) is not supported in TFX.

Note that not all terms can be used as formulae. Tuples, numbers, and "distinct objects" cannot be used as formulae.

3.2 Tuples

Tuples in TFX are written in `[]` brackets, and can contain any type of term, including formulae and variables of type `$o`. Signatures can contain tuple types. The following is the relevant BNF excerpt.

```
<tfx_tuple_type>       ::= [<tff_type_list>]
<tff_type_list>        ::= <tff_top_level_type> |
                           <tff_top_level_type>,<tff_type_list>
<tfx_tuple>            ::= [] | [<tff_arguments>]
<tff_arguments>        ::= <tff_term> | <tff_term>,<tff_arguments>
```

The tuple type $(\mathbb{R}, \mathbb{R})$ can be written in the TFX syntax as `[$real,$real]` and the type of addition for complex numbers $(\mathbb{R}, \mathbb{R}) \times (\mathbb{R}, \mathbb{R}) \rightarrow (\mathbb{R}, \mathbb{R})$ can be written as `([$real,$real] * [$real,$real]) > [$real,$real]`. The tuple term $(2, 3)$ can be written as `[2,3]`. Tuples can occur only as non-boolean terms (i.e., they cannot be formulae), anywhere they are well-typed. In the following example the predicate `p` takes a tuple $(\mathbb{Z}, \iota, o)$ as the first argument.

```
tff(p_type,type,p: ([$int,$i,$o] * $o * $int) > $o ).
tff(q_type,type,q: ($int * $i) > $o ).
tff(me_type,type,me: $i ).
tff(tuples_1,axiom,
    ! [X: $int] : p([33,me,$true],! [Y: $i] : q(X,Y),27) ).
```

While product types and tuple types are semantically equivalent, the two separate syntaxes make it easy to distinguish between the following cases.

```
tff(n_type,type,n: [$int,$int]).
tff(f_type,type,f: [$int,$int] > $int).
tff(g_type,type,g: ($int * $int) > $int).
tff(h_type,type,h: ([$int,$int] * $int) > $int).
```

The first case defines `n` to be a tuple of two integers. The second case defines `f` to be a function from a tuple of two integers to an integer. The third case defines `g` to be a function from two integers to an integer. The last case defines `h` to be a function from a tuple of two integers and an integer, to an integer.

The tuple syntax cannot be used to simultaneously declare types of multiple constants in an annotated formula with the `type` role. For example, the following expression is not valid.

```
tff(ab_type,type,[a,b]: [$int,$int]).
```

Instead, one must declare the type of each constant separately.

```
tff(a_type,type,a: $int).
tff(b_type,type,b: $int).
```

3.3 Conditional Expressions

Conditional expressions are polymorphic, taking a formula as the first argument, then two formulae or terms of the same type as the second and third arguments. The type of the conditional expression is the type of its second and third arguments. The following is the relevant BNF excerpt.

```
<tfx_conditional>      ::= $ite(<tff_logic_formula>,<tff_term>,<tff_term>)
```

The keyword `$ite` is used for conditional expressions occurring both as terms and formulae, which is different from the old TFF syntax of if-then-else that contained two separate keywords `$ite_t` and `$ite_f`.

The definition and a property of the *max* function in Formulae 4 and 5 can be expressed in TFX as

```
tff(max_type,type,max: ($int * $int) > $int).
tff(max_defn,axiom,
    ! [X: $int,Y: $int]: max(X,Y) = $ite($greatereq(X,Y),X,Y) ).
tff(max_property,conjecture,
    ! [X: $int,Y: $int]:
      $ite(max(X,Y) = X,$greatereq(X,Y),$greatereq(Y,X)) ).
```

3.4 Let Expressions

Let expressions in TFX contain (i) the type signatures of locally defined symbols; (ii) the definitions of the symbols; and (iii) the term or formula in which the definitions are used. The syntax of type signatures in let expressions is the same as for top-level type declrations. The definitions specify how the locally defined symbols are expanded in the term or formulae where they are used. The type signature must include the types for all the local defined symbols. The following is the relevant BNF excerpt.

```
<tfx_let>             ::= $let(<tfx_let_types>,<tfx_let_defns>,<tff_term>)
<tfx_let_types>       ::= <tff_atom_typing> | [<tff_atom_typing_list>]
<tff_atom_typing_list> ::= <tff_atom_typing> |
                           <tff_atom_typing>,<tff_atom_typing_list>
<tfx_let_defns>       ::= <tfx_let_defn> | [<tfx_let_defn_list>]
<tfx_let_defn>        ::= <tfx_let_LHS> <assignment> <tff_term>
<tfx_let_LHS>         ::= <tff_plain_atomic> | <tfx_tuple>
<tfx_let_defn_list>   ::= <tfx_let_defn> | <tfx_let_defn>,<tfx_let_defn_list>
```

The keyword `$let` is used for let expressions defining both function and predicate symbols, regardless of whether the let expression occurs as a term or a formula. This is different from the old TFF syntax of let expressions that contained four separate keywords `$let_tt`, `$let_tf`, `$let_ft`, and `$let_ff`.

In the following example an integer constant `c` is defined in a let expression.

```
tff(p_type,type,p: ($int * $int) > $o ).
tff(let_1,axiom,$let(c: $int,c:= $sum(2,3),p(c,c)) ).
```

The left hand side of a definition may contain pairwise distinct variables as top-level arguments of the defined symbol, and the variables can (typically do) also appear in the right hand side of the definition. Such variables are implicitly universally quantified, and are of the type defined by the symbol's type signature. The variables' values are supplied by unification in the defined symbol's use. For example, the let expression for the maximum of three integers in Formula 4 can be expressed in TFX as

```
tff(a_type,type,a: $int).
tff(b_type,type,b: $int).
tff(c_type,type,c: $int).
tff(p_type,type,p: $int > $o ).
tff(max_max,axiom,
    $let(max: ($int * $int) > $int,
         max(X,Y):= $ite($greatereq(X,Y),X,Y),
         p(max(max(a,b),c)) ) ).
```

and the let expression for the equivalence of two boolean constants in Formula 7 can be expressed in TFX as

```
tff(p,type,p: $o).
tff(q,type,q: $o).
tff(p_eq_q,axiom,
    $let(imply: ($o * $o) > $o,
         imply(X,Y):= ~X | Y,
         imply(p,q) & imply(q,p)) ).
```

Let expressions can use definitions of tuples. Formula 8 can be written in TFX as follows. Notice that the let expression's type declarations contain the elements of both tuples in the simultaneous definition.

```
tff(plus,type,
    plus: ([$real,$real] * [$real,$real]) > [$real,$real]).
tff(plus_def,axiom,
    ! [X: [$real,$real],Y: [$real$,$real]] :
      ( plus(X,Y)
      = $let([a: $real,b: $real,c: $real,d: $real],
             [[a,b]:= X, [c,d]:= Y],
             [$sum(a,c),$sum(b,d)]) ).
```

Sequential let expressions (`let*`) can be implemented by nesting let expressions. In the following example `ff` and `gg` are defined in sequence, and the let expression is equivalent to `p(f(i,i,i,i))`.

```
tff(i_type,type,i: $int).
tff(f_type,type,f: ($int * $int * $int * $int) > $int).
tff(p_type,type,p: $int > $o ).
tff(let_tuple_3,axiom,
    $let(ff: ($int * $int) > $int,
         ff(X,Y):= f(X,X,Y,Y),
         $let(gg: $int > $int,
              gg(Z):= ff(Z,Z),
              p(gg(i)) ) ) ).
```

Let expressions can have simultaneous local definitions with the type declarations and the definitions given in `[]`s (they look like tuples of declarations and definitions, but are specified independently of tuples in the syntax). The symbols must have distinct signatures. For example, the let expression to swap two constants in Formula 9 can be expressed in TFX as

```
tff(a,type,a: $i).
tff(b,type,b: $i).
tff(p,type,p: ($i * $i) > $o).
tff(pba,axiom,
    $let([a: $i,b: $i],
         [a:= b, b:= a],
         p(a,b))).
```

and the equivalent let expression using a tuple in Formula 10 can be expressed in TFX as

```
tff(a,type,a: $i).
tff(b,type,b: $i).
tff(p,type,p: ($i * $i) > $o).
tff(pba,axiom,
    $let([a: $i,b: $i],
         [a,b]:= [b,a],
         p(a,b))).
```

In the following example two function symbols are defined simultaneously, and the let expression is equivalent to `p(f(i,i,f(i,i,i,i),f(i,i,i,i)))`.

```
tff(i_type,type,i: $int).
tff(f_type,type,f: ($int * $int * $int * $int) > $int).
tff(p_type,type,p: $int > $o ).
tff(let_tuple_2,axiom,
    $let([ff: ($int * $int) > $int, gg: $int > $int],
         [ff(X,Y):= f(X,X,Y,Y), gg(Z):= f(Z,Z,Z,Z)],
         p(ff(i,gg(i)))) ).
```

The defined symbols of a let expression have scope over the formula or term in which the definitions are applied, shadowing any definition outside the let expression. The right hand side of a definition can have symbols with the same name as the defined symbol, but refer to symbols defined outside the let expression. In the following example the local definition of the `array` function symbol shadows the global declaration.

```
tff(array_type,type,array: $int > $real).
tff(p_type,type,p: $real > $o).
tff(let_3,axiom,
    $let(array: $int > $real,
         array(I):= $ite(I = 3,5.2,array(I)),
         p($sum(array(2),array(3))) ) ).
```

4 Software Support and Examples

4.1 Software for TFX

The BNF that defines TFX (and all the TPTP languages) provides the basis for the `BNFParser` family of automatically generated lex/yacc parsers for TPTP files. The parsers are available through the SystemB4TPTP online interface at http://www.tptp.org/cgi-bin/SystemB4TPTP. At the time of writing this paper, the TPTP4X utility is being upgraded to support TFX.

The Vampire theorem prover [9] supports all features of FOOL. Vampire transforms FOOL formulae into a set of first-order clauses using the VCNF

algorithm [6], and then reasons with these clauses using its usual resolution calculi for first-order logic. At the time of writing this paper the latest released version of Vampire, 4.2.2, uses a syntax for FOOL that differs slightly from TFX. Full support for the TFX syntax has been implemented in a recent revision of the Vampire source code[3], and will be available in the next release of Vampire.

TFX has been used by the program verification tools BLT [3] and Voogie [8]. BLT and Voogie read programs written in a subset of the Boogie intermediate verification language and generate their partial correctness properties written in the TFX syntax. BLT and Voogie generate formulae differently, but both rely on features of FOOL, namely conditional expressions, let expressions, and tuples.

4.2 Examples

Figures 3–5 show longer TFX examples using features of FOOL. Figure 3 shows how tuples, conditional expressions, and let expressions can be mixed, here to place two integer values in descending order as arguments in an atom. Figure 4 shows the TFX encoding of the FOOL formula in Figure 2, which expresses a partial correctness property of an imperative program with an `if` statement. Figure 5 shows an example that uses formulae as terms, in the second arguments of the `says` predicate. The problem is to find a model from which it is possible to determine which of `a`, `b`, or `c` is the only truthteller on this Smullyanesque island [10]. More TFX examples are available from the TPTP web site http://www.tptp.org/TPTP/Proposals/TFXExamples.tgz.

```
%----------------------------------------------------------------------
tff(v1_type,type,v1: $int).
tff(v2_type,type,v2: $int).
tff(ordered_p,axiom,
    $let([large: $int,small: $int],
         [large,small]:= $ite($greater(v1,v2),[v1,v2],[v2,v1]),
         p(large,small)) ).
%----------------------------------------------------------------------
```

Fig. 3. Mixing tuples, conditional and let expressions

5 Conclusion

This paper has introduced the eXtended Typed First-order form (TFX) of the TPTP's TFF language. TFX includes boolean variables as formulae, formulae as terms, tuple types and terms, conditional expressions, and let expressions.

[3] https://github.com/vprover/vampire

```
%----------------------------------------------------------------------
tff(x,type,x:$int).
tff(y,type,y:$int).
tff(t,type,t:$int).
tff(x_leq_y,conjecture,
    $let([x: $int,y: $int,t: $int],
         [x,y,t] := $ite($greater(x,y),
                         $let(t: $int,
                              t := x,
                              $let(x: $int,
                                   x := y,
                                   $let(y:$int,
                                        y := t,
                                        [x,y,t]))),
                         [x,y,t]),
         $lesseq(x,y))).
%----------------------------------------------------------------------
```

Fig. 4. A TFX encoding of the program analysis problem in Figure 2

```
%----------------------------------------------------------------------
tff(a_type,type, a: $i ).
tff(b_type,type, b: $i ).
tff(c_type,type, c: $i ).
tff(exactly_one_truthteller_type,type, exactly_one_truthteller: $o ).
tff(says,type, says: ( $i * $o ) > $o ).

%----Each person is either a truthteller or a liar
tff(island,axiom, ! [P: $i] : ( says(P,$true) <~> says(P,$false) ) ).
tff(exactly_one_truthteller,axiom,
    ( exactly_one_truthteller
  <=> ( ? [P: $i] : says(P,$true)
      & ! [P1: $i,P2: $i] :
          ( ( says(P1,$true) & says(P2,$true) )
         => P1 = P2 ) ) )).

%----B said that A said that there is exactly one truthteller
tff(b_says,hypothesis, says(b,says(a,exactly_one_truthteller)) ).

%----C said that what B said is false
tff(c_says,hypothesis, says(c,says(b,$false)) ).
%----------------------------------------------------------------------
```

Fig. 5. Who is the truthteller?

TFX is useful for (at least) concisely expressing problems coming from program analysis, and translated from more expressive logics.

Now that the syntax is settled, ATP system developers will be able to implement the new language features. It is already apparent from the SMT community that these are useful features, and systems that can already parse and reason using the SMT version 2 language need only new parsers to implement the features of TFX. In parallel, version v8.0.0 of the TPTP will include problems that use TFX, and the automated reasoning community is invited to submit problems for inclusion in the TPTP.

Acknowledgements. Thanks to our friends in the TPTP World who have provided feedback on TFX features, starting from the TPTP Tea Party at CADE-22 in 2009. The second author was partially supported by the Wallenberg Academy Fellowship 2014 and the Swedish VR grant D0497701.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (2010)
2. Blanchette, J., Paskevich, A.: TFF1: The TPTP Typed First-order Form with Rank-1 Polymorphism. In: Bonacina, M. (ed.) Proceedings of the 24th International Conference on Automated Deduction. pp. 414–420. No. 7898 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2013)
3. Chen, Y., Furia, C.: Triggerless Happy — Intermediate Verification with a First-Order Prover. In: Polikarpova, N., Schneider, S. (eds.) Proceedings of the 13th International Conference on Integrated Formal Methods. pp. 295–311. No. 10510 in Lecture Notes in Computer Science, Springer-Verlag (2017)
4. Kaliszyk, C., Sutcliffe, G., Rabe, F.: TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) Proceedings of the 5th Workshop on the Practical Aspects of Automated Reasoning. pp. 41–55. No. 1635 in CEUR Workshop Proceedings (2016)
5. Kotelnikov, E., Kovács, L., Reger, G., Voronkov, A.: The Vampire and the FOOL. In: Avigad, J., Chlipala, A. (eds.) Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs. pp. 37–48. ACM (2016)
6. Kotelnikov, E., Kovács, L., Suda, M., Voronkov, A.: A Clausal Normal Form Translation for FOOL. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) Proceedings of the 2nd Global Conference on Artificial Intelligence. pp. 53–71. No. 41 in EPiC Series in Computing (2016)
7. Kotelnikov, E., Kovács, L., Voronkov, A.: A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Proceedings of the International Conference on Intelligent Computer Mathematics. pp. 71–86. No. 9150 in Lecture Notes in Computer Science, Springer-Verlag (2015)
8. Kotelnikov, E., Kovács, L., Voronkov, A.: A FOOLish Encoding of the Next State Relations of Imperative Programs. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) Proceedings of the 9th International Joint Conference on Automated Reasoning (2018)

9. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: Sharygina, N., Veith, H. (eds.) Proceedings of the 25th International Conference on Computer Aided Verification. pp. 1–35. No. 8044 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2013)
10. Smullyan, R.: What is the Name of This Book? The Riddle of Dracula and Other Logical Puzzles. Prentice-Hall (1978)
11. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning **43**(4), 337–362 (2009)
12. Sutcliffe, G.: The TPTP World — Infrastructure for Automated Reasoning. In: Clarke, E., Voronkov, A. (eds.) Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 1–12. No. 6355 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2010)
13. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
14. Sutcliffe, G., Benzmüller, C.: Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. Journal of Formalized Reasoning **3**(1), 1–27 (2010)
15. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP Typed First-order Form with Arithmetic. In: Bjørner, N., Voronkov, A. (eds.) Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 406–419. No. 7180 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2012)
16. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning **21**(2), 177–203 (1998)