Representing Logics of Theorem Provers

TLTP: Tens of Logics for Theorem Provers

Fulya Horozal and Florian Rabe

Jacobs University Bremen, Germany

Abstract. The TPTP language has been very successful as an interchange format for theorem proving problems in first-order logic. Recently, several efforts have been made to extend it to other logics. But these extensions raise two principal concerns. Firstly, they require a substantially more complex type system than first-order logic. Secondly, it becomes necessary to specify the relations between the various TPTP logics.
We offer a solution by supplementing the TPTP language with a logical framework based on the LF framework. Using this framework, we can give concise formal definitions of the TPTP logics and relations between them. This provides out-of-the-box support for type-checking individual TPTP problems against the intended target logic as well as for automated theory translations between logics. The framework is extensible, and future TPTP extensions can be added easily; it also becomes possible to define combinations of existing extensions.
While our presentation is targeted at the TPTP family of logics, our approach extends to most logics used in theorem proving.

1 Introduction

TPTP [22] was introduced as a simple representation format for benchmark problems for automated theorem provers (ATPs) in first-order logic (FOL). It has been very successful at combining [13], applying [5], and evaluating [15] ATPs. De facto, it has become the standard input language for first-order theorem provers with most provers supporting it natively. The TPTP problem library comprises about 20000 problems from about 50 domains and comes with extensive tool support [20]. TPTP has also been used as a knowledge representation format for other large libraries [23,14].

While TPTP originally handled only classical unsorted FOL, it has gradually expanded to a variety of logics. These include in particular typing [2,21], polymorphism [3], higher-order types [2], and arithmetic [21]; a modal version was proposed in [18]. These extensions have been defined by extending the TPTP syntax. Moreover, semantic variations of existing logics can be formed by using the same syntax with a different semantics. Semantic variants include in particular intuitionistic variants of TPTP logics [19] and different choices of axioms (e.g., extensionality or non-empty types) in higher-order logics [1]. We expect the future interest in additional extensions to rise further as theorem provers are tackling more and more complex languages. Extensions that have been suggested or are already under development include, for example, product types, dependent types, and description and choice operators.

Thus, TPTP is a very promising candidate for a universal interface language for ATP developers and users. Such a language would permit the smooth integration of ATP (problem-solving) systems and (problem-generating) applications. As long as we only work with classical untyped first-order logic, this is already possible. But in general, the successful communication between ATP systems crucially depends on a common understanding of the semantics. Because different ATP system make different logical assumptions, which are often implicit in the implementation or only documented informally, the growing number of logics that are of interest to ATP systems present new challenges for an interface language.

Firstly, as FOL is a context-free language with a standardized abstract syntax and semantics, it was reasonable to focus on standardizing the concrete syntax. But for many other logics, this is not sufficient anymore. For example, the TPTP extensions [21] and [3] for typed logics must carefully elaborate on the intended context-sensitive syntax and semantics, and [2] even avoids fixing the semantics of the higher-order extension entirely. This is particularly worrisome in the case of semantic variants where the same syntax has multiple plausible meanings.

Secondly, it is not always obvious what the relations between different TPTP logics are. For example, there are intuitive sublanguage relations between untyped FOL, typed FOL, and higher-order logic. But these can be difficult to specify precisely, especially when the larger language introduces new concepts and then recovers the smaller language as a special case. Moreover, certain extensions should be defined as modules so that they can be combined flexibly. For example, arithmetic should be combinable with any typed logic, and the extension of FOL with polymorphism should be consistent with a future extension of higher-order with polymorphism.

We propose a solution to these problems by coupling TPTP with a logical framework. Thus, we can give concise, fully formal, human- and machine-readable definitions of the syntax and semantics of all TPTP logics.

Our framework is based on the logical framework LF [8,16], which has already been used in [2] to define the syntax of the higher-order logic of TPTP. To define and relate the different logics, we also use the module system for LF [17]. However, this is not enough. We also need our recent extension of LF [10] with declaration patterns. While LF focuses on defining the logical symbols, declaration patterns permit characterizing the legal declarations of non-logical symbols, a feature that proves crucial for our purposes.

Using this framework, we can give formalizations of the TPTP logics and logic morphisms between them. Concretely, we give representations of the syntax, the declarations patterns, and the semantics of FOL, typed FOL, polymorphic FOL, and HOL as well as their extensions with arithmetic. For the semantics, we choose a proof theoretic semantics, i.e., a system of inference rules. A model theoretical encoding as in [11] is also possible but more complex.

Moreover, our framework makes it possible to define new TPTP logics by combining existing features. We exemplify this by proposing a new TPTP logic by combining polymorphism and higher-order logic.

Our framework is supported by the Twelf [16] implementation of LF. Twelf provides a generic type checking and type inference algorithm. Therefore, TPTP files, which are often read or written by humans, can omit a lot of inferable types and arguments: Using Twelf, these can be inferred to generate type-enriched versions of TPTP problems, which are easily machine-readable.

To be self-contained, we give a brief overview of the used logical framework in Sect. 2. Then we give the representations of the TPTP logics and their relations in Sect. 3 and 4, respectively. In Sect. 5, we describe the integration between our logical framework and the TPTP syntax. We conclude in Sect. 6.

2 The Logical Framework

The framework we use for our representations is an extension of modular LF [8,17] with declaration patterns. Here we will briefly introduce our framework and direct the reader for the details of the framework to [10]. Below we give the fragment of the our grammar that is sufficient for this paper, where the parts pertaining to declaration patterns are given in gray:

Modules	M	::=	$\%\texttt{sig}\ T = \{\Sigma\} \mid \%\texttt{view}\ v : T_1 \to T_2 = \{\sigma\}$
Signatures	Σ	::=	$c : E \mid \%\texttt{include}\ T \mid \%\texttt{pattern}\ p = P$
Views	σ	::=	$c := E \mid \%\texttt{include}\ v \mid \%\texttt{pattern}\ p := P$
Expressions	E	::=	$\texttt{type} \mid c \mid x \mid \{x : E\}\, E \mid [x : E]\, E \mid E\, E$
		|	$E, E \mid [E]_{i:[1..E]} \mid E_E \mid \mathit{Nat} \mid 0 \mid \mathit{succ}(E)$
Patterns	P	::=	$p \mid \{\Sigma\} \mid [x : E]\, P \mid P\, E$

Modular LF Let us first consider the language without declaration patterns. Modules are the toplevel declarations. Their semantics is defined in terms of the category of LF *signatures* and signature morphism (see, e.g., [9]). We will call the latter *views*.

A non-modular signature Σ declares a list of typed constants c. Correspondingly, views from a signature T_1 to a signature T_2 consist of assignments $c := E$, which map T_1-constants to T_2-expressions.

Expressions are formed from the universe of types `type`, constants c, bound variables x, dependent function types (Π-types) $\{x : E\}\, E$, λ-abstraction $[x : E]\, E$, and application $E\, E$. As usual, we write $E_1 \to E_2$ instead of $\{x : E_1\}\, E_2$ if x does not occur in E_2. A valid view extends homomorphically to a (type-preserving) map of all T_1-expressions to T_2-expressions.

$\%\texttt{sig}\ \mathit{Forms} = \{$
  $\texttt{\$o} : \texttt{type}$
  $\vdash : \texttt{\$o} \to \texttt{type}$
  $\& : \texttt{\$o} \to \texttt{\$o} \to \texttt{\$o}$
  $\vdots$
  $\%\texttt{pattern}\ \mathit{axiom} = [F : \texttt{\$o}]\ \{$
    $m : \vdash F$
  $\}$
$\}$

To this, the module adds the ability for signatures and views to *include* other signatures and views, respectively.

As an example, consider the declaration of the signature *Forms*, which we will use for our representations in Sect. 3. It declares an LF-type $\$o$ of propositions and an $\$o$-indexed type family $\vdash$. This type family exemplifies how logic encodings in LF follow the Curry-Howard correspondence to represent judgments as types and proofs as terms: Terms of type $\vdash F$ represent derivations of the judgment "F is true". Furthermore, *Forms* declares all propositional connectives, among which we give & as an example. It also has one declaration pattern *axiom*, which we explain below.

Declaration Patterns Let us now consider the extension of modular LF with declaration patterns. Declaration patterns formalize what it means to be an arbitrary theory of a logic L: A declaration pattern gives a formal specification of the syntactic shape of a class of L-declarations, and in a legal L-theory, each declaration must match one of the L-patterns.

Declaration patterns P are formed from pattern constants p, signatures $\{\Sigma\}$, λ-abstractions $[x : E]\, P$, and applications of patterns P to expressions E.

For example, the declaration pattern *axiom* in *Forms* formalizes the shape of axiom declarations. Each axiom declaration is of the form $m : \vdash F$ for some proposition F. Such a declaration matches the pattern expression $\mathit{axiom}\, F$.

Even though most logics do not use sequences, it turns out that sequences are usually necessary to write down declaration patterns. For example, in theories of typed first-order logic, function symbol declarations use a sequence of types – the argument types of the function symbol. Therefore, our language also uses *expression sequences* and *natural numbers*. These are formed by the gray productions for expressions:

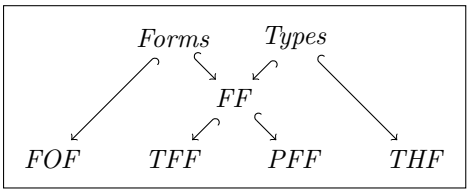
- E_1, E_2 for the concatenation of two sequences,
- E_n for the n-th element of E,
- $[E(x)]_{x:[1..n]}$ for the sequence $E(1), \ldots, E(n)$ where n has type Nat and $E(x)$ denotes an expression E with a free variable $x : \mathit{Nat}$; we write this sequence as E^n if x does not occur free in E,
- Nat, 0, and $\mathit{succ}(n)$ for the type of natural numbers.

We avoid giving the type system for this extension of LF and refer to [10] for the details. Intuitively, natural numbers and sequences occur only in pattern expressions, and fully applied pattern expressions normalize to expressions of the form $\{\Sigma\}$ where Σ is a plain LF signature. We will give examples below when we introduce specific declaration patterns.

A powerful feature of our sequences is that we can elegantly extend all left or right associative infix operators to sequences. For a sequence A of types that normalizes to $A_1, \ldots, A_n$ and for a type B, the type $A \to B$ normalizes to $A_1 \to \ldots \to A_n \to B$. Correspondingly, for a function f of that type and a sequence E that normalizes to $E_1, \ldots, E_n$, the expression $f\ E$ normalizes to $f\ E_1\ \ldots\ E_n$.

Finally, we also extend the module system to permit pattern assignments in views. The semantics of such a view from T_1 to T_2 is a functor mapping well-patterned T_1-theories to well-patterned T_2-theories.

3 Representing Logics



3.1 Syntactic Variants

In this section, we represent the TPTP languages in our logical framework. Specifically, we present the untyped first-order (*FOF*), the typed first-order (*TFF*), the polymorphic first-order (*PFF*) and the typed higher-order (*THF*) languages of TPTP. These form a diagram of LF-signatures as shown above, where $\hookrightarrow$ denotes inclusion. Where compatible with Twelf's concrete syntax, we will use the same symbol names as TPTP.

Untyped First-Order Language The signature *FOF* is given on the right. It includes the auxiliary signature *Forms* from Sect. 2 and adds the LF-type $\texttt{\$i} : \texttt{type}$ for the universe of first-order individuals. Moreover, it declares the first-order universal (!) and existential (?) quantifiers using higher-order abstract syntax, and the binary predicate symbol == for equality of individuals.

FOF contains two declaration patterns, *fun* and *pred*. These allow n-ary function and predicate symbols in *FOF*-theories, respectively. Recall that here $\texttt{\$i}^n$ abbreviates the sequence $\texttt{\$i}, \ldots, \texttt{\$i}$ of length n and that $(\texttt{\$i}, \ldots, \texttt{\$i}) \to \texttt{\$i}$ normalizes to $\texttt{\$i} \to \ldots \to \texttt{\$i} \to \texttt{\$i}$. This includes the case $n = 0$ of constant declarations. *FOF* additionally has the declaration pattern *axiom*, which is inherited from *Forms*.

```
%sig FOF = {
  %include Forms
  $i  : type
  !   : ($i → $o) → $o
  ?   : ($i → $o) → $o
  ==  : $i → $i → $o

  %pattern fun = [n : Nat] {
    f : $i^n → $i
  }
  %pattern pred = [n : Nat] {
    p : $i^n → $o
  }
}
```

Typed First-Order Languages To maximize reuse, we use two additional auxiliary LF-signatures, *Types* and *FF*, which contain the respective shared components of the typed first-order languages of TPTP.

```
%sig Types = {
  $tType : type
  $i     : $tType
  $tm    : $tType → type
}
```

Types is a base signature for all the typed TPTP languages. It declares an LF-type $\texttt{\$tType}$ that represents the universe of all TPTP types. It also declares a distinguished base type $\texttt{\$i} : \texttt{\$tType}$. We also use an LF-type family $\texttt{\$tm}$, which is an artifact of our Church-style, intrinsically typed representation and does not have an analog in TPTP. $\texttt{\$tm}$ assigns to each TPTP-type A an LF-type $\texttt{\$tm}\,A$ which contains the TPTP-terms of A. For example, the TPTP-terms of type $\texttt{\$i}$ are represented as LF-terms of LF-type $\texttt{\$tm}\,\texttt{\$i}$.

FF contains all the shared components of *TFF* and *PFF*. Besides typing and propositions, which are included from *Types* and *Forms*, respectively, it declares the logical symbols that are polymorphic over all TPTP types. These are the typed quantifiers ! (universal) and ? (existential) and typed equality ==. These cannot be declared in *Forms*, because they take a type argument $A : \texttt{\$tType}$. Note that we make use of the type inference capabilities of LF/Twelf here by making A an implicit argument that is automatically inferred from the context.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{FF}\ = \{ \\
\quad \%\texttt{include}\ \mathit{Types} \\
\quad \%\texttt{include}\ \mathit{Forms} \\
\quad ! \quad : (\$\texttt{tm}\, A \to \$\texttt{o}) \to \$\texttt{o} \\
\quad ? \quad : (\$\texttt{tm}\, A \to \$\texttt{o}) \to \$\texttt{o} \\
\quad == : \$\texttt{tm}\, A \to \$\texttt{tm}\, A \to \$\texttt{o} \\
\}
\end{array}
$$

We extend *FF* to obtain the languages *TFF* and *PFF*. *FF* already declares all logical symbols of *TFF* so that we only have to add the three declaration patterns:

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{TFF}\ = \{ \\
\quad \%\texttt{include}\ \mathit{FF} \\
\quad \%\texttt{pattern}\ \mathit{baseType}\ = \ \{ \\
\qquad t : \$\texttt{tType} \\
\quad \} \\
\quad \%\texttt{pattern}\ \mathit{typedFun}\ = \ [n : \mathit{Nat}]\,[A : \$\texttt{tType}^n]\,[B : \$\texttt{tType}]\ \{ \\
\qquad f : [\$\texttt{tm}\, A_i]_{i:[1..n]} \to \$\texttt{tm}\, B \\
\quad \} \\
\quad \%\texttt{pattern}\ \mathit{typedPred}\ = \ [n : \mathit{Nat}]\,[A : \$\texttt{tType}^n]\ \{ \\
\qquad p : [\$\texttt{tm}\, A_i]_{i:[1..n]} \to \$\texttt{o} \\
\quad \} \\
\}
\end{array}
$$

These patterns specify the form of the declarations of non-logical symbols that are allowed in *TFF*-theories:

- *baseType* allows the declaration of *TFF*-types t,
- *typedFun* allows the declaration of typed function symbols f that take arguments of *TFF*-type $A_1, \ldots, A_n$ and return an expression of type B,
- *typedPred* allows the declaration of typed predicate symbols p with arguments of *TFF*-types $A_1, \ldots, A_n$.

Note that our representation of *TFF* uses an LF-type $\$\texttt{o} : \texttt{type}$ in order to distinguish formulas from terms. This is different from the description in [21], where a TPTP-type $\$\texttt{o} : \$\texttt{tType}$ is used. Our representation has the advantage that we do not need case distinctions in order to avoid $\$\texttt{o}$ as an argument of a function or predicate symbol or of a quantifier.

Example 1 (TFF-Theories). Assume that a base type $\mathit{nat} : \$\texttt{tType}$ has already been declared (using the pattern *baseType*). Then the declaration of a binary function symbol on *nat* matches the pattern expression $\mathit{typedFun}\, 2\, (\mathit{nat}, \mathit{nat})\, \mathit{nat}$. The latter β-reduces to $\{f : [\$\texttt{tm}\, (\mathit{nat}, \mathit{nat})_i]_{i:[1..2]} \to \$\texttt{tm}\, \mathit{nat}\}$, which can be

simplified to $\{f : (\$\mathtt{tm}\,(nat, nat)_1, \$\mathtt{tm}\,(nat, nat)_2) \to \$\mathtt{tm}\,nat\}$ and eventually normalizes to $\{f : \$\mathtt{tm}\,nat \to \$\mathtt{tm}\,nat \to \$\mathtt{tm}\,nat\}$.

Interestingly, the logical symbols of *PFF* are almost the same as those of *TFF*. It only adds the universal ($!^\circ$) and existential ($?^\circ$) quantifiers over types. In the TPTP syntax, these are identified with ! and ?, but in LF their types are different so that they must be distinguished.

The crucial difference between the representations of *TFF* and *PFF* is in the legal declarations: *PFF*-theories may declare n-ary type operators and polymorphic function and predicate symbols. This shows the importance of declaration patterns in our framework as this difference could not be captured in plain LF.

$$
\begin{array}{l}
\%\mathtt{sig}\ \mathit{PFF}\ =\ \{ \\
\quad \%\mathtt{include}\ \mathit{FF} \\
\quad !^\circ : (\$\mathtt{tType} \to \$\mathtt{o}) \to \$\mathtt{o} \\
\quad ?^\circ : (\$\mathtt{tType} \to \$\mathtt{o}) \to \$\mathtt{o} \\
\quad \%\mathtt{pattern}\ \mathit{typeOp}\ =\ [n : \mathit{Nat}]\ \{ \\
\qquad t : \$\mathtt{tType}^n \to \$\mathtt{tType} \\
\quad \} \\
\quad \%\mathtt{pattern}\ \mathit{polyFun}\ =\ [m : \mathit{Nat}]\,[n : \mathit{Nat}]\,[A : (\$\mathtt{tType}^m \to \$\mathtt{tType})^n] \\
\qquad\qquad\qquad\qquad [B : \$\mathtt{tType}^m \to \$\mathtt{tType}]\,\{ \\
\qquad f : \{a : \$\mathtt{tType}^m\}\,[\$\mathtt{tm}\,(A_i\ a)]_{i:[1..n]} \to \$\mathtt{tm}\,(B\ a) \\
\quad \} \\
\quad \%\mathtt{pattern}\ \mathit{polyPred}\ =\ [m : \mathit{Nat}]\,[n : \mathit{Nat}]\,[A : (\$\mathtt{tType}^m \to \$\mathtt{tType})^n]\ \{ \\
\qquad p : \{a : \$\mathtt{tType}^m\}\,[\$\mathtt{tm}\,(A_i\ a)]_{i:[1..n]} \to \$\mathtt{o} \\
\quad \} \\
\}
\end{array}
$$

The pattern *typeOp* in *PFF* describes type operators t of arity n.

polyFun describes polymorphic function symbols f, which take m type arguments $a_1, \ldots, a_m$ and then n term arguments of types $A_1(a_1, \ldots, a_m), \ldots, A_n(a_1, \ldots, a_m)$ and return an expression of type $B(a_1, \ldots, a_m)$. Note that we use LF's higher-order abstract syntax to represent expressions of type $A : \$\mathtt{tType}$ with m free variables of type $\$\mathtt{tType}$ as terms of type $\$\mathtt{tType}^m \to \$\mathtt{tType}$.

Finally, *polyPred* describes polymorphic predicate symbols p, which take m type arguments $a_1, \ldots, a_m$ and then n term arguments of types $A_1(a_1, \ldots, a_m), \ldots, A_n(a_1, \ldots, a_m)$.

Note that via the inclusion of *FF*, *PFF* inherits the declaration pattern *axiom* of *Forms* that allows axioms of LF-type $\vdash F$ for some *PFF*-formula F.

Example 2 (PFF-Theories). Consider a unary type operator *list* has already been declared (using the pattern expression *typeOp* 1). Then the declaration of the *cons* operation on lists matches the pattern expression

$$\mathit{polyFun}\ 1\ 2\ \big(([a : \$\mathtt{tType}^1]\ a),\ ([a : \$\mathtt{tType}^1]\ \mathit{list}\ a)\big)\ ([a : \$\mathtt{tType}^1]\ \mathit{list}\ a)$$

which normalizes to $\{f : \{a : \$\mathtt{tType}\}\ a \to \mathit{list}\ a \to \mathit{list}\ a\}$.

Higher-Order Language *THF* is based on the one in [2]. Like *TFF* and *PFF*, it is based on the signature *Types*. It adds the logical symbols $>$ for function type formation, $\hat{}$ for λ-abstraction, and @ for application. As usual, we will write $>$ and @ as right and left-associative infix operators, respectively.

THF is not based on the signature *Forms*, which introduced the LF-type $\texttt{\$o} : \texttt{type}$ of formulas. Instead, it treats formulas as terms using a TPTP type $\texttt{\$o} : \texttt{\$tType}$. Consequently, the logical connectives and quantifiers and the truth judgment are declared based on $\texttt{\$o}$. Here we give only some example declarations.

Finally, *THF* uses three declaration patterns:

- *baseType* allows the declaration of *THF*-base types t,
- *typedCon* allows the declaration of typed constants c of type A for some *THF*-type A,
- *axiom* allows the declaration of axioms F for some *THF*-formula F.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{THF}\ = \{ \\
\quad \%\texttt{include}\ \mathit{Types} \\
\quad > : \texttt{\$tType} \to \texttt{\$tType} \to \texttt{\$tType} \\
\quad \hat{} \ : (\texttt{\$tm}\, A \to \texttt{\$tm}\, B) \to \texttt{\$tm}\, (A > B) \\
\quad @ \ : \texttt{\$tm}\, (A > B) \to \texttt{\$tm}\, A \to \texttt{\$tm}\, B \\
\quad \texttt{\$o} : \texttt{\$tType} \\
\quad \& \ : \texttt{\$tm}\, (\texttt{\$o} > \texttt{\$o} > \texttt{\$o}) \\
\quad ! \ : \texttt{\$tm}\, ((A > \texttt{\$o}) > \texttt{\$o}) \\
\quad \vdots \\
\quad \vdash \ : \texttt{\$tm}\, \texttt{\$o} \to \texttt{type} \\
\quad \%\texttt{pattern}\ \mathit{baseType}\ = \ \{ \\
\qquad t : \texttt{\$tType} \\
\quad \} \\
\quad \%\texttt{pattern}\ \mathit{typedCon}\ = \ [A : \texttt{\$tType}]\ \{ \\
\qquad c : \texttt{\$tm}\, A \\
\quad \} \\
\quad \%\texttt{pattern}\ \mathit{axiom}\ = \ [F : \texttt{\$tm}\, \texttt{\$o}]\ \{ \\
\qquad m : \vdash F \\
\quad \} \\
\}
\end{array}
$$

Using the declaration patterns in *THF*, we are able to define the theories of *THF* precisely. This is important because a number of different definitions are plausible. For example, the original higher-order logic of [4] arises if we drop the declaration pattern *baseType*. Another option is to use the pattern

$$\%\texttt{pattern}\ \mathit{neBaseType}\ = \ \{\, t : \texttt{\$tType},\ \mathit{nonempty} : \vdash\, ?\, @\, (\hat{}\, [x : \texttt{\$tm}\, t]\, \texttt{\$true})\,\}$$

so that every base type t is nonempty. Type definitions in the style of [7] can be obtained similarly.

Additional Features There is a variety of further syntactic variants, which can be seen as orthogonal features that can be added to a logic on demand. These include product types, conditional terms, let-expressions, etc. We find it important to separate each of them into its own module in order to permit a fine-grained control over the strength of a logic. Akin to [6], we call this the *little logics* approach. We will only give one example here and refer to [12] for further modules.

To add arithmetic as described in [21], we extend *TFF* with arithmetic operations in the signature *TFF-Arith* below. We only give a representative fragment of the encoding. Arithmetic domains are added as elements of the type $\texttt{\$adom}$,

and $\texttt{\$atype}$ includes the arithmetic domains into the universe $\texttt{\$tType}$ of types. This indirection is useful to quantify over exactly the arithmetic domains: It permits declaring the polymorphic operations $\texttt{\$sum}$, $\texttt{\$less}$, etc. for an arbitrary arithmetic domain D.

Twelf does not support arithmetic literals in the same way as TPTP. Therefore, we simply use a dummy constant $\texttt{\$lit}$, which represents arbitrary literals, e.g., all integer literals are written as $\texttt{\$lit}\ \texttt{\$int}$. As long as TPTP does not use dependent types, this is sufficient for all practical purposes, because identifying literals of the same type does not affect type checking.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{TFF\text{-}Arith}\ =\ \{ \\
\quad \%\texttt{include}\ \mathit{TFF} \\
\quad \texttt{\$adom}\ : \texttt{type} \\
\quad \texttt{\$atype} : \texttt{\$adom} \to \texttt{\$tType} \\
\quad \texttt{\$int}\ \ : \texttt{\$adom} \\
\quad \texttt{\$rat}\ \ : \texttt{\$adom} \\
\quad \texttt{\$real}\ : \texttt{\$adom} \\
\quad \texttt{\$sum}\ \ : \texttt{\$tm}\,(\texttt{\$atype}\,D) \to \texttt{\$tm}\,(\texttt{\$atype}\,D) \to \texttt{\$tm}\,(\texttt{\$atype}\,D) \\
\quad \texttt{\$less}\ : \texttt{\$tm}\,(\texttt{\$atype}\,D) \to \texttt{\$tm}\,(\texttt{\$atype}\,D) \to \texttt{\$o} \\
\quad \texttt{\$lit}\ \ : \{D : \texttt{\$adom}\}\,\texttt{\$tm}\,(\texttt{\$atype}\,D) \\
\quad \vdots \\
\}
\end{array}
$$

3.2 Semantic Variants

Since our framework is based on LF, we can concisely formalize the (proof-theoretical) semantics of the TPTP logics by giving LF signatures that represent the respective natural deduction calculus. While this is the first time such formalizations are given for all the TPTP logics, these are straightforward and well-known in principle. Therefore, we will omit the actual LF signatures and refer to our encodings in [12]. Instead, we focus on how to use them to distinguish semantic variants of the TPTP logics.

For the first-order logics, only one such distinction is of major importance: the one between classical and intuitionistic logic. Therefore, we use intuitionistic calculi for these logics and factor out the axiom of excluded middle into a separate module *ExclMid*.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{ExclMid}\ =\ \{ \\
\quad \%\texttt{include}\ \mathit{Forms} \\
\quad \mathit{em} : \vdash (A \mid (\sim A)) \\
\}
\end{array}
$$

If *IFOF*, *ITFF*, and *IPFF* are the signatures containing the respective intuitionistic calculus, we can define *CFOF*, *CTFF*, and *CPFF* for the respective classical variants, e.g., as on the right.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{CFOF}\ =\ \{ \\
\quad \%\texttt{include}\ \mathit{IFOF} \\
\quad \%\texttt{include}\ \mathit{ExclMid} \\
\}
\end{array}
$$

The situation is more complicated for the numerous semantic variants of higher-order logic. As a guiding principle, we make the base semantics of THF as weak as possible and define stronger logics by adding axioms and rules. Therefore, we define the signature *MinHOL* for the minimal proof theory of HOL. It

assumes λ-abstraction and equality as the only primitives. The former is axiomatized by β-conversion, and the latter by the rules for congruence relations. All further rules are added in separate signatures that extend *MinHOL*. In particular, these include:

Signature name	Added axioms/rules
PropExt	$(\vdash F \to \vdash G) \to (\vdash G \to \vdash F) \to \vdash F == G$
Xi	$\big(\{x : \mathtt{\$tm}\, A\} \vdash (S\, x) == (T\, x)\big) \to \vdash \hat{}\,[x]\,(S\, x) == \hat{}\,[x]\,(T\, x)$
FuncExt	$\big(\{x : \mathtt{\$tm}\, A\} \vdash (S \mathbin{@} x) == (T \mathbin{@} x)\big) \to \vdash S == T$
Eta	$\vdash \hat{}\,[x]\,(F \mathbin{@} x) == F$
BaseHOL	*PropExt*, *Xi*
BoolExt	*BaseHOL*, $\vdash F\, \mathtt{\$true} \to \vdash F\, \mathtt{\$false} \to \vdash\, ! \mathbin{@} F$
ExclMidHOL	*BaseHOL*, $\vdash\, \mid \mathbin{@} F \mathbin{@} (\sim \mathbin{@} F)$
NonEmptyTypes	*BaseHOL*, $\vdash\, ? \mathbin{@} (\hat{}\,[x : \mathtt{\$tm}\, A]\, \mathtt{\$true})$

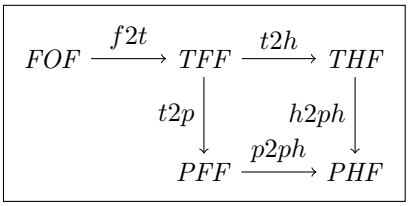
Intuitively, *PropExt* (propositional extensionality) identifies equality on booleans with logical equivalence; *Xi* provides the ξ-rule, a weak form of functional extensionality that can also be seen as a congruence rule for λ-abstraction; *Eta* provides η-conversion; *FuncExt* and *BoolExt* are functional and boolean extensionality; *ExclMidHOL* states excluded middle; and *NonEmptyTypes* makes all types non-empty.

By combining these and if necessary other extensions of *MinHOL*, we obtain the various incarnations of higher-order logics.

Of particular importance is the logic *BaseHOL*, which arises as the union of *MinHOL*, *Xi*, and *PropExt*. *BaseHOL* is strong enough to define all first-order connectives and quantifiers along with their (intuitionistic) natural deduction rules, and it is arguably the weakest combination of modules with that property. Therefore, we use it as a base logic for all *BoolExt*, *ExclMidHOL*, and *NonEmptyTypes*, which can only be formulated in the presence of logical connectives.

Note that *Eta* and *FuncExt* are equivalent, and so are *BoolExt* and *ExclMidHOL*. These relations can be formalized concisely as views between the respective signatures; these are given in [12].

4 Operations on Logics



We will now relate the logics from the previous section to each other using views and combine them to create a new logic *PHF*, resulting in the diagram on the right. Because each view induces a theory translation functor, this permits moving theories between the TPTP logics. We will focus on the most important views representing sublanguage relations; it is also possible to give (possibly partial) translations in the opposite directions, but there substantially more complicated to formalize.

Translating Logics The **view** $f2t$ **from** *FOF* **to** *TFF* is given below. Since *FOF* and *TFF* share the signature *Forms*, the view implicitly includes the identity translation of *Forms*.

$$
\begin{array}{l}
\%\texttt{view}\ f2t\ :\ \mathit{FOF}\ \to\ \mathit{TFF}\ =\ \{ \\
\quad \texttt{\$i}\ \ := \texttt{\$tm}\,\texttt{\$i} \\
\quad !\ \ := [f]\,!\,[x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x) \\
\quad ?\ \ := [f]\,?\,[x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x) \\
\quad ==\ := [x : \texttt{\$tm}\,\texttt{\$i}]\,[y : \texttt{\$tm}\,\texttt{\$i}]\,x\ ==\ y \\
\quad \mathit{fun}\ \ := [n : \mathit{Nat}]\,\{f : (\texttt{\$tm}\,\texttt{\$i})^n \to \texttt{\$tm}\,\texttt{\$i}\} \\
\quad \mathit{pred} := [n : \mathit{Nat}]\,\{p : (\texttt{\$tm}\,\texttt{\$i})^n \to \texttt{\$o}\} \\
\}
\end{array}
$$

The main characteristic of the translation is that the individuals of *FOF* are interpreted as the individuals of *TFF* of the distinguished base type `$i`. This is expressed as an assignment of the type `$i` of *FOF*-individuals to the type `$tm $i` in *TFF*. Correspondingly, *FOF*-quantifiers and equality are interpreted as the *TFF*-quantifiers and equality on the type `$tm $i`. Therefore, we map, for instance, ! to the expression $[f]\,!\,[x : \texttt{\$tm}\ \texttt{\$i}]\,(f\,x)$ that takes a *TFF*-formula f with a free variable $x : \texttt{\$tm}\ \texttt{\$i}$ and returns the universally quantified *TFF*-formula $!\,[x : \texttt{\$tm}\ \texttt{\$i}]\,(f\,x)$.

For each *FOF*-pattern p, the pattern translation maps every *FOF* declaration that matches p to a *TFF* declaration. This is defined by two assignments to the *FOF*-patterns *fun* and *pred*. For instance, *fun* is mapped to the pattern expression $[n : \mathit{Nat}]\,\{f : (\texttt{\$tm}\,\texttt{\$i})^n \to \texttt{\$tm}\,\texttt{\$i}\}$ so that every n-ary *FOF*-function symbol declaration is translated to the *TFF*-declaration of an n-ary function on `$tm $i`.

Note that our framework enforces that all views preserve typing. For example, the *FOF*-symbol $==: \texttt{\$i} \to \texttt{\$i} \to \texttt{\$o}$ must be mapped to a *TFF*-expression of type $\texttt{\$tm}\,\texttt{\$i} \to \texttt{\$tm}\,\texttt{\$i} \to \texttt{\$o}$. Similarly, the *FOF*-pattern *fun*, which takes a natural number and returns a signature, must be mapped to a *TFF*-pattern expression, which takes a natural number and returns a signature.

We give the **view** $t2p$ **from** *TFF* **to** *PFF* below. Since *TFF* and *PFF* share *FF* and *TFF* only adds declaration patterns, the view only consists of declaration pattern assignments.

$$
\begin{array}{l}
\%\texttt{view}\ t2p\ :\ \mathit{TFF}\ \to\ \mathit{PFF}\ =\ \{ \\
\quad \mathit{baseType}\ \ := \mathit{typeOp}\,0 \\
\quad \mathit{typedFun}\ \ := [n : \mathit{Nat}]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,\mathit{polyFun}\,0\,n\,A\,B \\
\quad \mathit{typedPred} := [n : \mathit{Nat}]\,[A : \texttt{\$tType}^n]\,\mathit{polyPred}\,0\,n\,A \\
\}
\end{array}
$$

Every *TFF*-type is interpreted as a nullary type operator in *PFF*. This is given as an assignment of the pattern *baseType* of *TFF* to the pattern *typeOp* supplied with 0 as the argument for the number of type arguments. Note that β-reducing $\mathit{typeOp}\,0$ results in the pattern expression $\{t : \texttt{\$tType}^0 \to \texttt{\$tType}\}$, where $\texttt{\$tType}^0$ normalizes to the empty sequence so that the whole type normalizes to `$tType`.

Every n-ary typed function symbol of *TFF* is interpreted as an n-ary polymorphic function symbol that does not take type arguments. This is given as an assignment from the pattern *typedFun* to the pattern expression

$[n : Nat]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,polyFun\,0\,n\,A\,B$, which takes the arity n of the function symbol, the sequence A of argument types, and the return type B and returns the corresponding monomorphic *PFF*-declaration. Note that after η-contraction, this pattern expression is equal to $polyPred\,0$. The pattern *typedPred* is translated accordingly.

Example 3 (Translating Theories). Consider a *TFF*-theory T containing the two declarations from Ex. 1. Applying the view $t2p$ to it yields a *PFF*-theory T'. Due to the assignment to *baseType*, $t2p$ translates the T-type *nat* to a T'-type of the same name. Due to the assignment to *typedFun*, $t2p$ translates the pattern expression $typedFun\,2\,(nat, nat)\,nat$ to

$$\big([n : Nat]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,polyFun\,0\,n\,A\,B\big)\,2\,(nat, nat)\,nat$$

which simplifies to $\{f : \{x : \texttt{type}^0\}\,\texttt{\$tm}\,nat \to \texttt{\$tm}\,nat \to \texttt{\$tm}\,nat\}$. Here $\texttt{\$tType}^0$ normalizes to the empty sequence of types so that the binding $\{x : \texttt{type}^0\}$ binds no variables and disappears, yielding the expected declaration.

The **view** $t2h$ **from** *TFF* **to** *THF* interprets the *TFF* type $\texttt{\$o} : \texttt{type}$ in terms of the *THF* constant $\texttt{\$o} : \texttt{\$tType}$ and translates the connectives to their higher-order analogues. Function and predicate symbols declared in terms of $\to$ over *TFF* are translated to the according declarations in terms of $>$ over *THF*. The translation of axioms is straightforward.

$$\begin{array}{lll}
\%\texttt{view}\ t2h : TFF \to THF = \{ & & \\
\quad \texttt{\$o} & := \texttt{\$tm}\,\texttt{\$o} & \\
\quad \vdash & := [F : \texttt{\$tm}\,\texttt{\$o}] \vdash F & \\
\quad \& & := [A : \texttt{\$tm}\,\texttt{\$o}]\,[B : \texttt{\$tm}\,\texttt{\$o}]\,\&\,@\,A\,@\,B & \\
\quad ! & := [f : \texttt{\$tm}\,\texttt{\$i} \to \texttt{\$tm}\,\texttt{\$o}]\,!\,@\,(\hat{}\,f) & \\
\quad \vdots & & \\
\quad typedFun & := [n : Nat]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,\{f' : \texttt{\$tm}\,(A > B)\} & \\
\quad typedPred & := [n : Nat]\,[A : \texttt{\$tType}^n]\,\{p' : \texttt{\$tm}\,(A > \texttt{\$o})\} & \\
\quad axiom & := [F : \texttt{\$tm}\,\texttt{\$o}]\,axiom\,F & \\
\} & &
\end{array}$$

Combining Logics A particular strength of a logical framework like ours is the ability to combine logics using colimits as studied in [9]. In the simplest case, this is just taking the union of two logics as we already did in Sect. 3.2. More generally, we can use pushouts in the category of LF signatures. We will give two interesting examples how our framework guides the design of new TPTP logics.

Firstly, we obtain *THF-Arith*, the extension of *THF* with arithmetic, by applying the theory translation functor induced by the view $t2h$. It maps the *TFF*-theory *TFF-Arith* to the corresponding *THF*-theory. This construction is obtained automatically from our framework and results in the commuting diagram above.

$$\begin{array}{ccc}
TFF & \xrightarrow{t2h} & THF \\
\big\downarrow & & \big\downarrow \\
TFF\text{-}Arith & \longrightarrow & THF\text{-}Arith
\end{array}$$

Secondly, we combine *PFF* and *THF* into a new logic: polymorphic higher-order logic *PHF*. This construction uses the commuting diagram below. If we ignore the patterns and consider only the underlying LF signatures, this diagram is obtained automatically as a pushout. However, we have to add the patterns of *PHF* – which merge the patterns of *PHF* and *THF* in a non-trivial way – manually. The relevant fragments of the LF signature for *PHF* is given below where THF' represents *THF* without its patterns. We omit the straightforward views *p2ph* and *h2ph*.

$$\begin{array}{ccc} TFF & \xrightarrow{t2h} & THF \\ {\scriptstyle t2p}\downarrow & & \downarrow{\scriptstyle h2ph} \\ PFF & \xrightarrow[p2ph]{} & PHF \end{array}$$

$$\begin{array}{l}
\%\texttt{sig}\ PHF\ =\ \{ \\
\quad \%\texttt{include}\ THF' \\
\quad !^{\circ} : (\$\texttt{tType} \to \$\texttt{tm}\,\$\texttt{o}) \to \$\texttt{tm}\,\$\texttt{o} \\
\quad ?^{\circ} : (\$\texttt{tType} \to \$\texttt{tm}\,\$\texttt{o}) \to \$\texttt{tm}\,\$\texttt{o} \\
\quad \%\texttt{pattern}\ typeOp\ =\ [n : Nat]\ \{ \\
\qquad t : \$\texttt{tType}^{n} \to \$\texttt{tType} \\
\quad \} \\
\quad \%\texttt{pattern}\ typedPolyCon\ =\ [m : Nat]\,[A : \$\texttt{tType}^{m} \to \$\texttt{tType}]\ \{ \\
\qquad c : \{a : \$\texttt{tType}^{m}\}\,\$\texttt{tm}\,(A\,a) \\
\quad \} \\
\}
\end{array}$$

5 Integration with TPTP

Multiple Logics in TPTP We suggest defining the TPTP logics in a suite of LF signatures, similar to the one we provided, and to maintain them in some official location, e.g., `http://www.tptp.org/Logics`. Moreover, we suggest adding a field `logic` to the header of a TPTP theory, whose value is a list of strings. If this field has the value `L1 ... Ln`, its meaning is that the theory is formed over the union of the logics `L1`, ..., `Ln`. This field can be used by problem authors and system implementers to determine whether an ATP system is applicable to a specific problem.

Most of the time this list will have length 1. By permitting a union of theories, users can form semantic variants of a logic by choosing a particular combination of rules. For example, a theorem of classical extensional higher-order logic can be characterized by `logic: BaseHOL Eta ExclMidHOL`.

Translating TPTP to LF In order to validate TPTP problems using Twelf, we give a translation from TPTP theories to LF signatures, which generalizes the existing translation for THF [2] to the case of arbitrary logics. As we have used LF identifiers with appropriate names and fixities, most of the syntactic idiosyncrasies of TPTP are captured exactly in LF. Therefore, the translation is straightforward. The details are given in the appendix.

Validating TPTP Theories Using the above translation, we can type-check TPTP theories in our logical framework. Then we can define a TPTP theory to be **strictly valid** if its translation is a well-formed signature of LF

with declaration patterns. In particular, strict validity implies that all declarations i) type-check, i.e., each declaration is a well-formed LF declaration, and ii) pattern-check, i.e., each declaration conforms to one of the patterns of the included logics.

This definition does not take Twelf type inference into account yet. We define a TPTP theory to be **loosely valid** if its translation is a well-formed signature of Twelf with declaration patterns. This means that Twelf type reconstruction can uniquely find the corresponding strictly valid signature. In fact, Twelf type reconstruction is much stronger than that. For example, in the polymorphic first-order language, we can omit essentially all type arguments because Twelf's implicit argument reconstruction can infer them.

6 Conclusion

We have applied results from the area of logical frameworks to automated theorem proving. By formalizing various logics used in theorem provers in a logical framework, it becomes possible to concisely define the language – including its semantics – supported by a theorem prover. Dually, users can specify the language required for a certain problem. This improves upon current interface languages, which have so far focused on specifying the syntax of the logics. We suggest using our approach to give normative reference definitions of the type systems and the semantics of the logics used by ATP systems.

In order to maximize the benefit of our approach, we have closely coupled it with the TPTP language, the quasi standard ATP interface language. Our work is designed to supplement TPTP without interfering with any established work flows. We have provided definitions for the current TPTP languages in our formal framework (available at [12]) and described how to use them to link individual ATP problems to their logic. Further logics can be defined easily. In particular, future logic definitions can reuse existing ones, and we have made use of this to introduce a TPTP logic for polymorphic higher-order logic.

Our approach can also provide novel implementation support: validating TPTP problems against a target logic, performing type reconstruction on them, and translating problems between logics. Moreover, if an ATP system can return a proof object as an LF-term, LF can act as a reference proof checker.

References

1. C. Benzmüller and C. Brown. A Structured Set of Higher-Order Problems. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 66–81. Springer, 2005.
2. C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 – The core of the TPTP Language for Higher-Order Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2008.
3. J. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. 2012. in preparation.

4. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
5. E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference*, pages 198–212. Springer, 2004.
6. W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.
7. M. Gordon and A. Pitts. The HOL Logic. In M. Gordon and T. Melham, editors, *Introduction to HOL, Part III*, pages 191–232. Cambridge University Press, 1993.
8. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
9. R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
10. F. Horozal. Logic translations with declaration patterns. https://svn.kwarc.info/repos/fhorozal/pubs/patterns.pdf, 2012.
11. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412(37):4919–4945, 2011.
12. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see https://trac.omdoc.org/LATIN/.
13. L. Paulson and K. Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.
14. A. Pease and G. Sutcliffe. First Order Reasoning on a Large Ontology. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Empirically Successful Automated Reasoning in Large Theories*, number 257 in CEUR Workshop Proceedings, pages 59–69, 2007.
15. F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.
16. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
17. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009.
18. T. Raths and J. Otten. Building a Problem Library for First-Order Modal Logics. In *TABLEAUX 2009 Position Papers and Workshop Proceedings*, 2009.
19. T. Raths, J. Otten, and C. Kreitz. The ILTP Problem Library for Intuitionistic Logic. *Journal of Automated Reasoning*, 38(1-3):261–271, 2007.
20. G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 1–12. Springer, 2010.
21. G. Sutcliffe, K. Claessen S. Schulz, and P. Baumgartner. The TPTP Typed First-order Form with Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 2012. to appear.
22. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
23. J. Urban. MPTP 0.2: Design, Implementation, and Initial Experiments. *Journal of Automated Reasoning*, 37(1-2):21–43, 2006.

A Translation from TPTP to LT

In this section, we give the technical details of the translation from TPTP syntax to LF/Twelf syntax.

Firstly, the non-trivial cases of the translation of formulas are the following (where we use `F'` for the translation of `F`):

- A TPTP application `c(F1,...,Fn)` is translated to `(c F1' ... Fn')` where `c` is any (possibly polymorphic) function or predicate symbol or a type operator.
- A quantified formula `Q[X1:A1,...,Xn:An]:(F)` is translated to `(Q[X1:$tm A1'] ... Q[Xn:$tm An']F')` where `Q` is either `!`, `?`, or `^`. Type quantifiers are translated accordingly but require additionally replacing the name of the quantifier. Quantifiers binding both term and type variables are translated accordingly.
- Equality `=` is translated to `==`.
- Arithmetic literals of domain D are translated to `$lit` D.

Thus, the translation of formulas is very simple and can even be implemented using a few regular expressions.

Secondly, a TPTP declaration `LAN(NAME,ROLE,F)` where `ROLE` is any assertion role such as `axiom` or `conjecture`, is translated to the LF declaration `NAME: $istrue (F')`. (Here `$istrue` is an alias for the symbol $\vdash$.)

A TPTP declaration `include('Axioms/THEORY.ax')` is translated to the LF declaration `%include ax.THEORY`, where `ax` is an LF namespace prefix bound to some URI reserved for TPTP axiom sets, e.g., `http://www.tptp.org/Axioms`.

Finally, a TPTP declaration `LAN(NAME,type,c:F)` is translated to `c:F'`. Here, the translation of `F` to `F'` depends on the TPTP language `LAN` – in fact, we only we need to distinguish between the first-order languages and the higher-order languages. In the first-order case, we use LF function types to represent TPTP function types. Therefore, type quantifiers `!>[A:$tType]:F` are translated to LF-Π-binders `{A:$tType}F'`. And the type operators `*` and `>` are both translated to Twelf's curried type operator `->`. In the higher-order case, the functions of higher-order logic can be used directly to represent function types, and `F'` is simply `$tm F`.

Thirdly, to translate a TPTP theory with name `N` and logic `L1 ... Ln`, we first add all omitted `type` declarations (if any). If the resulting theory contains the declarations $D_1, \ldots, D_n$, we translate it to the LF signature

%namespace $ax =$ "http://www.tptp.org/Axioms"
%namespace $log =$ "http://www.tptp.org/Logics"
%sig $N = \{$
%include $log.L1$
$\vdots$
%include $log.Ln$
D'_1
$\vdots$
D'_n
$\}$

where D' is the translation of D and log is some fixed namespace reserved for TPTP logics. Here we assume the namespace `http://www.tptp.org/Logics` to be the namespace reserved for the TPTP logics.