Formal Logic Definitions for Interchange Languages

Fulya Horozal and Florian Rabe

Jacobs University Bremen, Germany

Abstract. System integration often requires standardized interchange languages, via which systems can exchange mathematical knowledge. Major examples are the MathML-based markup languages and TPTP. However, these languages standardize only the syntax of the exchanged knowledge, which is insufficient when the involved logics are complex or numerous. Logical frameworks, on the other hand, allow representing the logics themselves (and are thus aware of the semantics), but they abstract from the concrete syntax.
Maybe surprisingly, until recently, state-of-the-art logical frameworks were not quite able to adequately represent logics commonly used in formal systems. Using a recent extension of the logical framework LF, we show how to give concise formal definitions of the logics used in TPTP. We can also formally define translations and combinations between the various TPTP logics. This allows us to build semantics-aware tool support such as type-checking TPTP content.
While our presentation focuses on the current TPTP logics, our approach can be easily extended to other logics and interchange languages. In particular, our logic representations can be used with both TPTP and MathML. Thus, a single definition of the semantics can be used with either interchange syntax.

1 Introduction

Interchange Languages System integration is one of the biggest challenges concerning formal systems. One major line of research has been the development of interchange languages that allow exchanging mathematical knowledge in standardized formats.

Two languages have been particularly successful: the MathML family (which we use here to group together MathML [1], OpenMath [4], and OMDoc [13]) developed in the CICM community and the TPTP family [21] developed in the deduction community

It may be surprising to some readers that we consider these two families to be very similar. Indeed, TPTP was originally introduced for benchmark problems for first-order logic (FOL) provers, whereas MathML is meant to support all mathematical content. But TPTP syntax is becoming more and more expressive, including typing [22], higher-order logic (HOL), [2], polymorphism [3] and arithmetic [22]. As a consequence, the two languages share a central property:

Both are primarily defined by an extremely expressive context-free grammar and use only informal descriptions of the fragment of well-formed, meaningful content. MathML is broader, allowing users to define and document well-formed fragments in content dictionaries. TPTP is deeper, slowly specifying individual fragments in an ongoing series of papers (which is barely keeping up with user demand for specifying more expressive languages – the most recent extension proposal [15] appears in the same volume as our present paper). But in both cases, a formal description (e.g., with a context-sensitive inference system) is lacking.

Maybe provocatively, we can say that both MathML and TPTP standardize the syntax but not the semantics. Because successful system integration usually requires a common understanding of the semantics, and because the number and complexity of formal systems is growing, this is becoming an important issue for interface languages.

Moreover, it is not always obvious what the relations are between the various well-formed fragments used in different applications. For example, there are intuitive sublanguage relations between TPTP's untyped FOL, typed FOL, and HOL. But these can be difficult to specify precisely, e.g., when the larger language introduces new concepts and then recovers the smaller language as a special case. Moreover, it is desirable to specify language fragments modularly 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 logic with polymorphism.

Logical Frameworks Logical frameworks [17] such as Twelf [18] or Isabelle [16], in some sense, suffer from the opposite problem: They allow very concise definitions of the well-formed fragments (often with tool-support to decide which expressions are meaningful) but have little-to-no bearing on the actual syntax used in communication between systems. In fact, logics specifications in logical frameworks tend to be idealized and not in sync with those implicitly used in the above interchange languages.

Moreover, state-of-the-art logical frameworks are surprisingly unable to fully specify standard logics: They allow specifying the well-formed objects but not the well-formed theories. For example, it is not possible to specify in LF that a first-order theory should not declare higher-order function symbols. Similarly, Isabelle cannot specify the shape of HOL type definitions. This problem was remedied recently in [10], which develops a logical framework that extends LF with declaration patterns [11], which specify the shape of declarations in well-formed theories.

Contribution We apply the framework of [10] to give concise, fully formal, human- and machine-readable specifications of a set of logics commonly used with interchange languages. We focus on the logics in the TPTP family, whose practical value is well-documented, but our approach is applicable to most other logics. Because our specifications are modular, we can combine features conveniently. We demonstrate this by defining TPTP's polymorphic HOL (which has

not been described previously) by taking a pushout of HOL and polymorphic FOL.

Our specifications are tightly integrated with both TPTP and MathML. Firstly, the framework's syntax can match the TPTP syntax almost exactly (the necessary minor transformation is maintained by Geoff Sutcliffe and the second author). In fact, our specifications have quasi-official status because they are now used by Sutcliffe to type-check the TPTP library. Secondly, the logical framework we use is implemented within MMT [20]. Thus, our specifications double as content dictionaries (which MMT internally maintains in OMDoc syntax), which MMT uses to check MathML content.

Thus MathML and TPTP become alternative concrete encodings, and a single logic specification defines well-formed MathML and TPTP content at the same time.

Overview We summarize the logical framework we use in Section 2. Then we specify the logics and their relations in Section 3 and 4, respectively. We describe implementation aspects in Section 5 and conclude in Section 6.

2 The Logical Framework

Technically, our logical framework arises in a rather complex way using four different features: We use *i)* the foundation-independent module system MMT [20] *ii)* extended with the declaration patterns of [10] and *iii)* instantiated with a version of LF [8] with *iv)* sequences [10,12]. We will refer to the resulting framework as LFS. Below we give the fragment of LFS that is sufficient for our purposes as a self-contained grammar in Figure 1:

Modules	M	$::=$	$\texttt{theory}\ T = \{\Sigma\} \mid \texttt{view}\ v : T_1 \to T_2 = \{\sigma\}$
Theories	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	$::=$	$c \mid x \mid \{x : E\}\, E \mid [x : E]\, E \mid E\, E$
		$\mid$	$\cdot \mid E, E \mid [E]_{i=1}^{E} \mid E_E \mid \texttt{nat} \mid 0 \mid succ(E) \mid \texttt{type}^E$
Patterns	P	$::=$	$p \mid \{\Sigma\} \mid [x : E]\, P \mid P\, E$

Fig. 1. Grammar

Module System Modules are the toplevel declarations. Their semantics is defined in terms of the category of MMT *theories* and theory morphisms. We call the latter *views*.

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

map T_1-constants to T_2-expressions. Views extends homomorphically to a (type-preserving) map of T_1-expressions to T_2-expressions.

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

LF Expressions are formed from 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.

As an example, consider the following declarations

```
theory Forms = {
  $o : type
  ⊢ : $o → type
  & : $o → $o → $o
  .
  :
  pattern axiom = [F : $o] {
    m : ⊢ F
  }
}
```

declaration of the theory *Forms*, which we will use for our representations in Section 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 Declaration patterns formalize the shape of well-formed theories of a logic L: In a well-formed L-theory, each declaration must match one of the L-patterns. For example, in a well-formed first-order theory, each symbol declaration must match one of these patterns: *i)* n-ary first-order function symbols, *ii)* n-ary first-order predicates symbols and *iii)* axioms.

But in LF, nothing stops users from writing theories that contain non-first-order declarations such as $f : (\texttt{\$i} \to \texttt{\$i}) \to \texttt{\$i}$ or $g : \texttt{\$o} \to \texttt{\$i}$ (where `$i` is the LF-type of first-order terms). These are still well-typed in LF even in the context of first-order logic. But in LFS, we can formalize the above three declaration patterns, and LFS will reject declarations that do not match one of the patterns.

Declaration patterns P are formed from pattern constants p, theories $\{\Sigma\}$, λ-abstractions $[x : E]\, P$, and applications of patterns P to expressions E. Further details on declaration patterns are given in [10].

Example 1. The declaration pattern *axiom* in *Forms* formalizes the shape of axiom declarations. These must be of the form $m : \vdash F$ for some proposition F, which matches the declaration pattern *axiom* F.

Adequacy Representations of logics in LF are well-known to be adequate. However, technically, that is only true for the representation of *expressions*. Representations in LF are not actually adequate with respect to *theories* because

the LF type theory cannot rule out declarations that do not match a pattern. The declaration patterns of LFS are exactly what is needed to obtain an adequate representation of the theories of a logic.

Moreover, for all logics L we consider, all well-formed L-theories in LFS simplify to theories in the LF fragment of LFS. Therefore, we can inherit the adequacy for expressions from LF.

Sequences 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 underlined productions for expressions:

- $\cdot$ for the empty sequence,
- 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 $x : \texttt{nat}$; we write this sequence as E^n if x does not occur free in E,
- `nat` for the type of natural numbers,
- 0 and $succ(n)$ for zero and the successor of a given natural number n,
- $\texttt{type}^n$ for the kind of a sequence expression of length n.

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 closed pattern expressions normalize to expressions of the form $\{\Sigma\}$ where Σ is an LF theory. We will give examples below when we introduce specific declaration patterns.

A powerful feature of our sequences is that we can elegantly extend the primitives of LF to flexary operators. In particular, 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 $(\ldots(f\ E_1)\ \ldots\ E_n)$.

Finally, we also extend views from T_1 to T_2 to map pattern constants to pattern expressions. The semantics of such a view is a functor mapping well-formed T_1-theories to well-formed T_2-theories. We will give examples when we introduce specific views.

3 Representing Logics

In this section, we represent the TPTP languages in our logical framework. Specifically, we present the untyped first-order (*FOF*), the typed first-order (*TF0*) and its extension with arithmetic (*TFA*), the polymorphic first-order (*TF1*) and the typed higher-order (*TH0*) languages of TPTP.

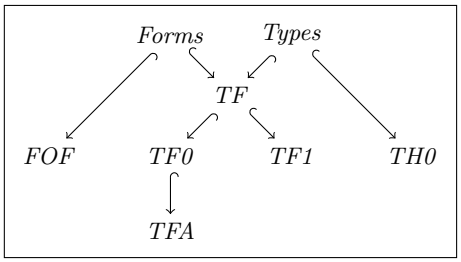

Fig. 2. TPTP Logics

Our representations form a diagram of LFS-theories as shown in Figure 2, 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 theory *FOF* is given on the right. It includes the auxiliary theory *Forms* from Section 2 and adds the LFS-type $\$\mathtt{i} : \mathtt{type}$ 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 $\$\mathtt{i}^n$ abbreviates the sequence $\$\mathtt{i}, \ldots, \$\mathtt{i}$ of length n and that $(\$\mathtt{i}, \ldots, \$\mathtt{i}) \to \$\mathtt{i}$ normalizes to $\$\mathtt{i} \to \ldots \to \$\mathtt{i} \to \$\mathtt{i}$. This includes the case $n = 0$ of constant declarations. *FOF* additionally has the declaration pattern *axiom*, which is inherited from *Forms*.

$$
\begin{array}{l}
\mathtt{theory}\ \mathit{FOF}\ =\ \{ \\
\quad \mathtt{include}\ \mathit{Forms} \\
\quad \$\mathtt{i}\ \ : \mathtt{type} \\
\quad !\ \ \ : (\$\mathtt{i} \to \$\mathtt{o}) \to \$\mathtt{o} \\
\quad ?\ \ \ : (\$\mathtt{i} \to \$\mathtt{o}) \to \$\mathtt{o} \\
\quad == : \$\mathtt{i} \to \$\mathtt{i} \to \$\mathtt{o} \\
\quad \mathtt{pattern}\ \mathit{fun}\ =\ [n : \mathtt{nat}]\ \{ \\
\qquad f : \$\mathtt{i}^n \to \$\mathtt{i} \\
\quad \} \\
\quad \mathtt{pattern}\ \mathit{pred}\ =\ [n : \mathtt{nat}]\ \{ \\
\qquad p : \$\mathtt{i}^n \to \$\mathtt{o} \\
\quad \} \\
\}
\end{array}
$$

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

$$
\begin{array}{l}
\mathtt{theory}\ \mathit{Types}\ =\ \{ \\
\quad \$\mathtt{tType} : \mathtt{type} \\
\quad \$\mathtt{i}\ \ \ \ \ : \$\mathtt{tType} \\
\quad \$\mathtt{tm}\ \ \ \ : \$\mathtt{tType} \to \mathtt{type} \\
\}
\end{array}
$$

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

example, the TPTP-terms of type $\texttt{\$i}$ are represented as LFS-terms of LFS-type $\texttt{\$tm}\,\texttt{\$i}$.

TF contains all the shared components of *TF0* and *TF1*. 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 here we make A an implicit argument in the style of Twelf that is automatically inferred from the context.

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

We extend *TF* to obtain the languages *TF0* and *TF1*. *TF* already declares all logical symbols of *TF0* so that we only have to add the three declaration patterns:

$$
\begin{array}{l}
\texttt{theory}\ \mathit{TF0}\ =\ \{ \\
\quad \texttt{include}\ \mathit{TF} \\
\quad \texttt{pattern}\ \mathit{baseType}\ =\ \{ \\
\qquad t : \texttt{\$tType} \\
\quad \} \\
\quad \texttt{pattern}\ \mathit{typedFun}\ =\ [n : \texttt{nat}]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\ \{ \\
\qquad f : [\texttt{\$tm}\,A_i]_{i=1}^n \to \texttt{\$tm}\,B \\
\quad \} \\
\quad \texttt{pattern}\ \mathit{typedPred}\ =\ [n : \texttt{nat}]\,[A : \texttt{\$tType}^n]\ \{ \\
\qquad p : [\texttt{\$tm}\,A_i]_{i=1}^n \to \texttt{\$o} \\
\quad \} \\
\}
\end{array}
$$

These patterns specify the form of the declarations of non-logical symbols that are allowed in *TF0*-theories:

- *baseType* allows the declaration of *TF0*-types t,
- *typedFun* allows the declaration of typed function symbols f that take arguments of *TF0*-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 *TF0*-types $A_1, \ldots, A_n$.

Note that our representation of *TF0* uses an LFS-type $\texttt{\$o} : \texttt{type}$ in order to distinguish formulas from terms. This is different from the description in [22], 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 2 (TF0-Theories). Assume that a base type $nat : \texttt{\$tType}$ has already been declared (using the pattern *baseType*). Then the declaration of a binary function symbol on *nat* matches the pattern expression $\mathit{typedFun}\,2\,(nat, nat)\,nat$.

The latter β-reduces to $\{f : [\$\mathtt{tm}\,(nat, nat)_i]_{i=1}^2 \to \$\mathtt{tm}\,nat\}$, which can be simplified to $\{f : \big(\$\mathtt{tm}\,(nat, nat)_1, \$\mathtt{tm}\,(nat, nat)_2\big) \to \$\mathtt{tm}\,nat\}$ and eventually normalizes to $\{f : \$\mathtt{tm}\,nat \to \$\mathtt{tm}\,nat \to \$\mathtt{tm}\,nat\}$.

Interestingly, the logical symbols of *TF1* are almost the same as those of *TF0*. It only adds the universal ($!^\circ$) and existential ($?^\circ$) quantifiers over types. In the TPTP syntax, these are identified with ! and ?, but in LFS their types are different so that they must be distinguished.

The crucial difference between the representations of *TF0* and *TF1* is in the legal declarations: *TF1*-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 LF.

$\mathtt{theory}\ \mathit{TF1}\ =\ \{$
$\quad\mathtt{include}\ \mathit{TF}$
$\quad!^\circ : (\$\mathtt{tType} \to \$\mathtt{o}) \to \$\mathtt{o}$
$\quad?^\circ : (\$\mathtt{tType} \to \$\mathtt{o}) \to \$\mathtt{o}$
$\quad\mathtt{pattern}\ \mathit{typeOp}\ =\ [n : \mathtt{nat}]\ \{$
$\qquad t : \$\mathtt{tType}^n \to \$\mathtt{tType}$
$\quad\}$
$\quad\mathtt{pattern}\ \mathit{polyFun}\ =\ [m : \mathtt{nat}]\,[n : \mathtt{nat}]\,[A : (\$\mathtt{tType}^m \to \$\mathtt{tType})^n]$
$\qquad\qquad [B : \$\mathtt{tType}^m \to \$\mathtt{tType}]\,\{$
$\qquad f : \{a : \$\mathtt{tType}^m\}\,[\$\mathtt{tm}\,(A_i\ a)]_{i=1}^n \to \$\mathtt{tm}\,(B\ a)$
$\quad\}$
$\quad\mathtt{pattern}\ \mathit{polyPred}\ =\ [m : \mathtt{nat}]\,[n : \mathtt{nat}]\,[A : (\$\mathtt{tType}^m \to \$\mathtt{tType})^n]\ \{$
$\qquad p : \{a : \$\mathtt{tType}^m\}\,[\$\mathtt{tm}\,(A_i\ a)]_{i=1}^n \to \$\mathtt{o}$
$\quad\}$
$\}$

The pattern *typeOp* in *TF1* 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 higher-order abstract syntax in the style of LF 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 *TF*, *TF1* inherits the declaration pattern *axiom* of *Forms* that allows axioms of LFS-type $\vdash F$ for some *TF1*-formula F.

Example 3 (TF1-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 *TH0* is based on the one in [2]. Like *TF0* and *TF1*, it is based on the theory *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.

TH0 is not based on the theory *Forms*, which introduced the LFS-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, *TH0* uses three declaration patterns:

- *baseType* allows the declaration of *TH0*-base types t,
- *typedCon* allows the declaration of typed constants c of type A for some *TH0*-type A,
- *axiom* allows the declaration of axioms F for some *TH0*-formula F.

Using the declaration patterns in *TH0*, we are able to define the theories of *TH0* precisely. This is important because a number of different definitions are plausible. For example, the original higher-order logic of [5] 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.

$$\begin{array}{l}
\texttt{theory}\ \mathit{TH0} = \{ \\
\quad \texttt{include}\ \mathit{Types} \\
\quad > : \texttt{\$tType} \to \texttt{\$tType} \to \texttt{\$tType} \\
\quad \hat{} : (\texttt{\$tm}\, A \to \texttt{\$tm}\, B) \to \texttt{\$tm}\, (A > B) \\
\quad @ : \texttt{\$tm}\, (A > B) \to \texttt{\$tm}\, A \to \texttt{\$tm}\, B \\
\quad \texttt{\$o} : \texttt{\$tType} \\
\quad \& : \texttt{\$tm}\, (\texttt{\$o} > \texttt{\$o} > \texttt{\$o}) \\
\quad ! : \texttt{\$tm}\, ((A > \texttt{\$o}) > \texttt{\$o}) \\
\quad \vdots \\
\quad \vdash : \texttt{\$tm}\, \texttt{\$o} \to \texttt{type} \\
\quad \texttt{pattern}\ \mathit{baseType} = \{ \\
\qquad t : \texttt{\$tType} \\
\quad \} \\
\quad \texttt{pattern}\ \mathit{typedCon} = [A : \texttt{\$tType}]\ \{ \\
\qquad c : \texttt{\$tm}\, A \\
\quad \} \\
\quad \texttt{pattern}\ \mathit{axiom} = [F : \texttt{\$tm}\, \texttt{\$o}]\ \{ \\
\qquad m : \vdash F \\
\quad \} \\
\}
\end{array}$$

Arithmetic To add arithmetic as described in [22], we extend *TF0* with arithmetic operations in the theory *TFA* 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.

```
theory TFA = {
  include TF0
  $adom  : type
  $atype : $adom → $tType
  $int   : $adom
  $rat   : $adom
  $real  : $adom
  $sum   : $tm ($atype D) → $tm ($atype D) → $tm ($atype D)
  $less  : $tm ($atype D) → $tm ($atype D) → $o
  ⋮
}
```

Other Syntactic 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, choice operators, conditional terms, let-expressions, etc. We separate these into individual modules to gain fine-grained control over the strength of a logic. Akin to [6], we call this the *little logics* approach.

Semantic Variants The above has specified *well-formed* formulas. But we can also specify the *valid* formulas by formalizing appropriate calculi. For each logical operator, we formalize natural deduction proof rules. These rules are straightforward, and we refer to our formalizations in [14].

We explicitly mention only some axioms, which are important because they distinguish semantic variants of the same syntax. As a guiding principle, we make the base calculi as weak as possible and define axioms in separate modules, which can be included on demand.

For first-order logics, only one semantic variant is of major importance: the one between classical and intuitionistic logic. Therefore, we use the module on the right for the axiom of excluded middle.

```
theory ExclMid = {
  include Forms
  em : ⊢ (A | (∼ A))
}
```

The situation is more complicated for higher-order logic. The theory *MinHOL* defines the minimal proof theory of HOL using only (i) β-conversion for λ-abstraction and (ii) congruence rules for equality. All further rules are added in separate modules that extend *MinHOL* as introduced in Figure 3: *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. *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 combines *MinHOL*, *Xi*, and *PropExt*: It is the weakest

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

Fig. 3. Modules Extending *MinHOL*

reasonable variant of HOL that is strong enough to define all first-order connectives and quantifiers and their (intuitionistic) natural deduction rules. We use it as a base logic for all *BoolExt*, *ExclMidHOL*, and *NonEmptyTypes*, which can only be formulated in the presence of some 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 [14].

4 Translating and Combining Logics

We will now relate the logics from the previous section to each other using views and combine them to create a new logic *TH1* (polymorphic higher-order logic), resulting in the diagram in Figure 4. 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; there are also (possibly partial) translations in the opposite directions, but they are substantially more complicated to formalize.

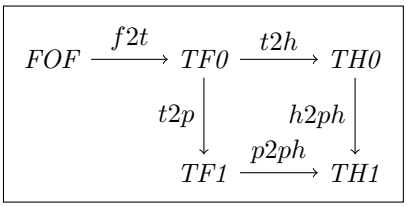

Fig. 4. Translating TPTP Logics

Notably, users working with implementations of higher-order logic have already started using ad hoc variants of *TH1* (independent of our work) in expectation of an eventual adoption as an official TPTP logic. This shows that our work offers an efficient way for TPTP to keep up with the growing demand for reference definitions of logics.

Translating Logics The **view** *f2t* **from** *FOF* **to** *TF0* is given below. Since *FOF* and *TF0* share the theory *Forms*, the view implicitly includes the identity translation of *Forms*.

The main characteristic of the translation is that the individuals of *FOF* are interpreted as the individuals of *TF0* of the distinguished base type $\texttt{\$i}$. This is expressed as an assignment of the type $\texttt{\$i}$ of *FOF*-individuals to the type $\texttt{\$tm}\,\texttt{\$i}$ in *TF0*. Correspondingly, *FOF*-quantifiers and equality are interpreted as the *TF0*-quantifiers and equality on the type $\texttt{\$tm}\,\texttt{\$i}$. Therefore, we map, for instance, ! to the expression $[f]\,![x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x)$ that takes a *TF0*-formula f with a free variable $x : \texttt{\$tm}\,\texttt{\$i}$ and returns the universally quantified *TF0*-formula $![x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x)$.

$$
\begin{array}{l}
\texttt{view}\ f2t\ :\ \mathit{FOF}\ \rightarrow\ \mathit{TF0}\ =\ \{\\
\quad \texttt{\$i}\ \ := \texttt{\$tm}\,\texttt{\$i}\\
\quad !\ \ := [f]\,![x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x)\\
\quad ?\ \ := [f]\,?[x : \texttt{\$tm}\,\texttt{\$i}]\,(f\,x)\\
\quad ==\ := [x : \texttt{\$tm}\,\texttt{\$i}]\,[y : \texttt{\$tm}\,\texttt{\$i}]\,x\ ==\ y\\
\quad \mathit{fun}\ := [n : \texttt{nat}]\,\{f : (\texttt{\$tm}\,\texttt{\$i})^n \rightarrow \texttt{\$tm}\,\texttt{\$i}\}\\
\quad \mathit{pred} := [n : \texttt{nat}]\,\{p : (\texttt{\$tm}\,\texttt{\$i})^n \rightarrow \texttt{\$o}\}\\
\}
\end{array}
$$

For each *FOF*-pattern p, the pattern translation maps every *FOF* declaration that matches p to a *TF0* declaration. This is defined by two assignments to the *FOF*-patterns *fun* and *pred*. For instance, *fun* is mapped to the pattern expression $[n : \texttt{nat}]\,\{f : (\texttt{\$tm}\,\texttt{\$i})^n \rightarrow \texttt{\$tm}\,\texttt{\$i}\}$ so that every n-ary *FOF*-function symbol declaration is translated to the *TF0*-declaration of an n-ary function on $\texttt{\$tm}\,\texttt{\$i}$.

Note that our framework enforces that all views preserve typing. For example, the *FOF*-symbol ==: $\texttt{\$i} \rightarrow \texttt{\$i} \rightarrow \texttt{\$o}$ must be mapped to a *TF0*-expression of type $\texttt{\$tm}\,\texttt{\$i} \rightarrow \texttt{\$tm}\,\texttt{\$i} \rightarrow \texttt{\$o}$. Similarly, the *FOF*-pattern *fun*, which takes a natural number and returns a theory, must be mapped to a *TF0*-pattern expression, which takes a natural number and returns a theory.

We give the **view** *t2p* **from** *TF0* **to** *TF1* below. Since *TF0* and *TF1* share *TF* and *TF0* only adds declaration patterns, the view only consists of declaration pattern assignments.

$$
\begin{array}{l}
\texttt{view}\ t2p\ :\ \mathit{TF0}\ \rightarrow\ \mathit{TF1}\ =\ \{\\
\quad \mathit{baseType}\ := \mathit{typeOp}\,0\\
\quad \mathit{typedFun}\ := [n : \texttt{nat}]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,\mathit{polyFun}\,0\,n\,A\,B\\
\quad \mathit{typedPred} := [n : \texttt{nat}]\,[A : \texttt{\$tType}^n]\,\mathit{polyPred}\,0\,n\,A\\
\}
\end{array}
$$

Every *TF0*-type is interpreted as a nullary type operator in *TF1*. This is given as an assignment of the pattern *baseType* of *TF0* 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 $\{t : \texttt{\$tType}^0 \rightarrow \texttt{\$tType}\}$, where $\texttt{\$tType}^0$ normalizes to the empty sequence so that the whole type normalizes to $\texttt{\$tType}$.

Every n-ary typed function symbol of *TF0* 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 $[n : \texttt{nat}]\,[A : \texttt{\$tType}^n]\,[B : \texttt{\$tType}]\,\mathit{polyFun}\,0\,n\,A\,B$, 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 *TF1*-declaration. Note that af-

ter η-contraction, this pattern expression is equal to $\mathit{polyPred}\,0$. The pattern *typedPred* is translated accordingly.

Example 4 (Translating Theories). Consider a *TF0*-theory T containing the two declarations from Ex. 2. Applying the view $t2p$ to it yields a *TF1*-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 $\mathit{typedFun}\,2\,(\mathit{nat},\mathit{nat})\,\mathit{nat}$ to

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

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

The **view** *t2h* **from** *TF0* **to** *TH0* interprets the *TF0* type $\mathtt{\$o} : \mathtt{type}$ in terms of the *TH0* constant $\mathtt{\$o} : \mathtt{\$tType}$ and translates the connectives to their higher-order analogues. Function and predicate symbols declared in terms of $\to$ over *TF0* are translated to the respective declarations in terms of $>$ over *TH0*. The translation of axioms is straightforward.

$$\begin{array}{lll}
\mathtt{view}\ t2h : \mathit{TF0} \to \mathit{TH0} = \{ & & \\
\quad \mathtt{\$o} & := \mathtt{\$tm}\,\mathtt{\$o} & \\
\quad \vdash & := [F:\mathtt{\$tm}\,\mathtt{\$o}] \vdash F & \\
\quad \& & := [A:\mathtt{\$tm}\,\mathtt{\$o}]\,[B:\mathtt{\$tm}\,\mathtt{\$o}]\,\&\,@\,A\,@\,B & \\
\quad ! & := [f:\mathtt{\$tm}\,\mathtt{\$i} \to \mathtt{\$tm}\,\mathtt{\$o}]\,!\,@\,(\hat{}\,f) & \\
\quad \vdots & & \\
\quad \mathit{typedFun} & := [n:\mathtt{nat}]\,[A:\mathtt{\$tType}^n]\,[B:\mathtt{\$tType}]\,\{f':\mathtt{\$tm}\,(A >^* B)\} & \\
\quad \mathit{typedPred} & := [n:\mathtt{nat}]\,[A:\mathtt{\$tType}^n]\,\{p':\mathtt{\$tm}\,(A >^* \mathtt{\$o})\} & \\
\quad \mathit{axiom} & := [F:\mathtt{\$tm}\,\mathtt{\$o}]\,\mathit{axiom}\,F & \\
\} & &
\end{array}$$

Note that $>^* : \mathtt{\$tType}^n \to \mathtt{\$tType} \to \mathtt{type}$ is the flexary operator derived from $>$ (in LFS, the flexary version of a binary operator is definable, see [12]).

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. More generally, we can use pushouts in the category of theories. We will give two interesting examples how our framework guides the design of new TPTP logics.

Firstly, we obtain *THA*, the extension of *TH0* with arithmetic, by applying the theory translation functor induced by the view *t2h*. It maps the *TF0*-theory *TFA* to the corresponding *TH0*-theory. This construction is obtained automatically from our framework and results in the commuting diagram in Figure 5.

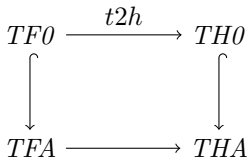

Fig. 5. TPTP Logics with Arithmetic

Secondly, we combine *TF1* and *TH0* into a new logic: polymorphic higher-order logic *TH1*. This construction uses the commuting diagram in Figure 6.

If we ignore the patterns and consider only the underlying LF-theories, this diagram is obtained automatically as a pushout. However, we have to add the patterns of *TH1* – which merge the patterns of *TH1* and *TH0* in a non-trivial way – manually. The relevant fragments of the LFS-theory for *TH1* is given below where *TH0′* represents *TH0* without its patterns. We omit the straightforward views *p2ph* and *h2ph*.

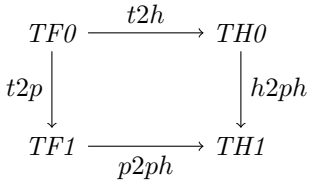

Fig. 6. Constructing *TH1*

```
theory TH1 = {
   include TH0′
   !° : ($tType → $tm $o) → $tm $o
   ?° : ($tType → $tm $o) → $tm $o

   pattern typeOp = [n : nat] {
      t : $tType^n → $tType
   }
   pattern typedPolyCon = [m : nat] [A : $tType^m → $tType] {
      c : {a : $tType^m} $tm (A a)
   }
}
```

5 Practical Aspects

Processing Content Declaration patterns are implemented as a part of Mmt [19]. Our specifications are written in Mmt instantiated with LF and sequences. Mmt can check the logic specifications and generates content dictionaries (in the form of OMDoc theories) from them. As the namespaces for these content dictionaries, we use URIs derived from `http://www.tptp.org/`.

If theories or objects of these logics are given in OMDoc/MathML syntax, they can be read and type-checked natively by Mmt.

If theories or objects are given in TPTP syntax, we use the fact that Mmt is closely integrated with the Twelf tool [18]. In particular, we obtain a Twelf signature for our logics. The TPTP distribution includes a converter from TPTP to Twelf syntax, and Sutcliffe uses Twelf to type-check TPTP content relative to this signature. (This works even though Twelf only supports LF and not sequences because sequences never occur in TPTP theories, only in the logic specifications.) Twelf in turn can export its input as OMDoc, which can be used for further processing by Mmt or other tools. Twelf can also act as a reference proof checker if systems produce proofs.

Logic Ascription in TPTP Content Mmt and OMDoc require content to reference the logic it is written in, but TPTP does not. Indeed, users can mix and match language features in the same TPTP file. Therefore, the above-mentioned

converter actually translates all content into the largest logic, i.e., polymorphic HOL with arithmetic.

We propose adding a value to the header of a TPTP theory, which is a list of strings and defines the target logic to be used during type checking. The meaning of the value `L1 ... Ln` would be that the theory is formed over the union of the logics `L1`, ..., `Ln`. Incidentally, it could be used by problem authors and system implementers to determine whether an ATP system is applicable to a specific problem. For example, this information could be included in the value of the existing `SPC` header field.

Our proposal is also the best solution to the problem of semantic variants: The status of a problem (i.e., whether it is a theorem) may depend on the chosen logic. So far, TPTP has side-stepped this issue because it mostly occurred in the form of intuitionistic vs. classical FOL. (The official TPTP policy is that intuitionistic provers are welcome but incomplete.) But with higher-order provers becoming more sophisticated, it is likely to become necessary to record which logic a problem is supposedly provable in.

6 Conclusion

We observed that interchange languages commonly used for system integration – like MathML or TPTP – focus on standardizing the context-free syntax. But they do not formalize the context-sensitive language fragments corresponding to the well-formed expressions of individual logics. By formalizing these logics in a logical framework, it becomes possible to concisely specify these fragments.

We systematically applied this approach to obtain a suite of formal specifications of logics commonly used in formal systems. We focused on the TPTP logics, the quasi standard for automated deduction systems, but further logics can be defined easily, possibly reusing existing ones. We applied this modular design to obtain a new TPTP-style logic for polymorphic higher-order logic.

Our specifications are both human- and machine-readable. And they are tightly integrated with the concrete syntax and tool support of the MathML and TPTP interchange languages, inducing type checkers and serving as content dictionaries. Therefore, we propose them as reference definitions of these logics. In fact, TPTP has effectively adopted our proposal already by using our specifications for type-checking.

References

1. R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, M. Froumentin, R. Hunter, P. Ion, M. Kohlhase, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt. Mathematical Markup Language (MathML) Version 2.0 (second edition), 2003. See http://www.w3.org/TR/MathML2.
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*, pages 491–506. Springer, 2008.

3. J. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In M. Bonacina, editor, *Automated Deduction*, pages 414–420. Springer, 2013.
4. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.
5. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
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. *A Framework for Defining Declarative Languages*. PhD thesis, Jacobs University Bremen, 2014.
11. F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.
12. F. Horozal, F. Rabe, and M. Kohlhase. Flexary Operators for Formalized Mathematics. In S. Watt, J. Davenport, A. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 312–327. Springer, 2014.
13. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.
14. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. see https://trac.omdoc.org/LATIN/.
15. 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, *Intelligent Computer Mathematics*. Springer, 2015.
16. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
17. F. Pfenning. Logical frameworks. In J. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.
18. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction*, pages 202–206, 1999.
19. F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.
20. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.
21. 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.
22. G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence*, pages 406–419. Springer, 2012.