THF0 – The core of the TPTP Language for Higher-Order Logic

Christoph Benzmüller[1]*, Florian Rabe[2], and Geoff Sutcliffe[3]

[1] Saarland University, Germany
[2] Jacobs University Bremen, Germany
[3] University of Miami, USA

Abstract. One of the keys to the success of the Thousands of Problems for Theorem Provers (TPTP) problem library and related infrastructure is the consistent use of the TPTP language. This paper introduces the core of the TPTP language for higher-order logic – THF0, based on Church's simple type theory. THF0 is a syntactically conservative extension of the untyped first-order TPTP language.

1 Introduction

There is a well established infrastructure that supports research, development, and deployment of first-order Automated Theorem Proving (ATP) systems, stemming from the Thousands of Problems for Theorem Provers (TPTP) problem library [29]. This infrastructure includes the problem library itself, the TPTP language [27], the SZS ontologies [30], the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries [26], and the CADE ATP System Competition (CASC) [28]. This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems.

One of the keys to the success of the TPTP and related infrastructure is the consistent use of the TPTP language. Until now the TPTP language has been defined for only untyped first-order logic (first-order form (FOF) and clause normal form (CNF)). This paper introduces the core of the TPTP language for classical higher-order logic – THF0, based on Church's simple type theory. THF0 is the *core* language in that it provides only the commonly used and accepted aspects of a higher-order logic language. The full THF language includes the THFF extension that allows the use of first-order style prefix syntax, and the THF1 and THF2 extensions that provide successively richer constructs in higher-order logic.[1]

* This work was supported by EPSRC grant EP/D070511/1 (LEO-II)

[1] The THFF, THF1, and THF2 extensions have already been completed. The inital release of only THF0 allows users to adopt the language without being swamped by the richness of the full THF language. The full THF language definition is available from the TPTP web site, www.tptp.org.

As with the first-order TPTP language, a common adoption of the THF language will enable convenient communication of higher-order data between different systems and researchers. THF0 is the starting point for building higher-order analogs of the existing first-order infrastructure components, including a higher-order extension to the TPTP problem library, evaluation of higher-order ATP systems in CASC, TPTP tools to process THF0 problems, and a higher-order TSTP proof representation format. An interesting capability that has become possible with the THF language is easy comparison of higher-order and first-order versions of problems, and evaluation of the relative benefits of the different encodings with respect to ATP systems for the logics.

2 Preliminaries

2.1 The TPTP Language

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language suitable for writing both ATP problems and solutions. The BNF of the THF0 language, which defines common TPTP language constructs and the THF0 specific constructs, is given in Appendix A.

The top level building blocks of the TPTP language are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

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

An example annotated first-order formula, supplied from a file, is:

```
fof(formula_27,axiom,
    ! [X,Y] :
      ( subclass(X,Y)
    <=> ! [U] :
          ( member(U,X)
         => member(U,Y) )),
    file('SET005+0.ax',subclass_defn),
    [description('Definition of subclass'), relevance(0.9)]).
```

The *language*s supported are first-order form (`fof`), clause normal form (`cnf`), and now typed higher-order form (`thf`). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula in an ATP system - see the BNF for the list of recognized roles. The details of the *formula* notation for connectives and quantifiers can be seen in the BNF. The forms of identifiers for uninterpreted functions, predicates, and variables follow Prolog conventions, i.e., functions and predicates start with a lowercase letter, variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a `$`, or are composed of non-alphanumeric characters. The basic logical connectives are `!`, `?`,`~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for $\forall$, $\exists$,$\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Quantified variables follow the quantifier in square brackets, with a colon to separate the quantification from the logical formula. The *source* is an optional term

describing where the formula came from, e.g., an input file or an inference. The *useful_info* is an optional list of terms from user applications.

An `include` directive may include an entire file, or may specify the names of the annotated formulae that are to be included from the file. Comments in the TPTP language extend from a `%` character to the end of the line, or may be block comments within `/* ...*/` bracketing.

2.2 Higher-order Logic

There are many quite different frameworks that fall under the general label "higher-order". The notion reaches back to Frege's original predicate calculus [11]. Inconsistencies in Frege's system, caused by the circularity of constructions such as "the set of all sets that do not contain themselves", made it clear that the expressivity of the language had to be restricted in some way. One line of development, which became the traditional route for mathematical logic, and which is not addressed further here, is the development of axiomatic first-order set theories, e.g. Zermelo-Fraenkel set theory [32].

Russell suggested using type hierarchies, and worked out ramified type theory. Church (inspired by work of Carnap) later introduced simple type theory [9], a higher-order framework built on his simply typed λ calculus, employing types to reduce expressivity and to remedy paradoxes and inconsistencies. Church's simple type theory was later extended and refined in various ways, e.g., by Martin Löf who added type universes, and by Girard and Reynolds who introduced type systems with polymorphism. This stimulated much further research in intuitionistic and constructive type theory.

Variants and extensions of Church's simple type theory have been the logic of choice for interactive proof assistants such as HOL4 [13], HOL Light [15], PVS [22], Isabelle/HOL [21], and OMEGA [25]. Church's simple type theory is also a common basis for higher-order ATP systems. The TPS system [1], which is based on a higher-order mating calculus, is a pioneering ATP system for Church's simple type theory. More recently developed higher-order ATP systems, based on extensional higher-order resolution, are LEO [5] and LEO-II [8]. Otter-λ [2] is an extension of the first-order system Otter to an untyped variant of Church's type theory. This common use of Church's simple type theory motivates using it as the starting point for THF0. For the remainder of this paper "higher-order" is therefore synonymous with Church's simple type theory [4].

A major application area of higher-order logic is hardware and software verification. Several interactive higher-order proof assistants put an emphasis on this. For example, the Isabelle/HOL system has been applied in the Verisoft project [12]. The formal proofs to be delivered in such a project require a large number of user interactions – a resource intensive task to be carried out by highly trained specialists. Often only a few of the interaction steps really require human ingenuity, and many of them could be avoided through better automation support. There are several barriers that hamper the application of automated higher-order ATP systems in this sense: (i) the available higher-order ATP systems are not yet optimized for this task; (ii) typically there are large syntax gaps between the

higher-order representation languages of the proof assistants, and the input languages of the higher-order ATP systems; (iii) the results of the higher-order ATP systems have to be correctly interpreted and (iv) their proof objects might need to be translated back. The development of a commonly accepted infrastructure, and increased automation in higher-order ATP, will benefit these applications and reduce user interaction costs. Another promising application area is knowledge based reasoning. Knowledge based projects such as Cyc [19] and SUMO [20] contain a significant fraction of higher-order constructs. Reasoning in and about these knowledge sources will benefit from improved higher-order ATP systems. A strong argument for adding higher-order ATP to this application area is its demand for natural and human consumable problem and solution representations, which are harder to achieve after translating higher-order content into less expressible frameworks such as first-order logic. Further application areas of higher-order logic include computer-supported mathematics [24], and reasoning within and about multimodal logics [6].

2.3 Church's Simple Type Theory

Church's simple type theory is based on the simply typed λ calculus.

The set of simple types is freely generated from basic types ι (written `$i` in THF0) and o (`$o` in THF0), and possibly further base types using the function type constructor $\rightarrow$ (`>` in THF0).

Higher-order terms are built up from simply typed variables (X_α), simply typed constants (c_α), λ-abstraction, and application. It is assumed that sufficiently many special logical constants are available, so that all other logical connectives can be defined. For example, it is assumed that $\neg_{o\rightarrow o}$, $\vee_{o\rightarrow o\rightarrow o}$, and $\Pi_{(\alpha\rightarrow o)\rightarrow o}$ (for all simple types α) are given. The semantics of these logical symbols is fixed according to their intuitive meaning.

Well-formed (simply typed) higher-order terms are defined simultaneously for all simple types α. Variables and constants of type α are well-formed terms of type α. Given a variable X of type α and a term T of type β, the abstraction term $\lambda X.T$ is well-formed and of type $\alpha \rightarrow \beta$. Given terms S and T of types $\alpha \rightarrow \beta$ and α respectively, the application term $(S\ T)$ is well-formed and of type β.

The initial target semantics for THF0 is Henkin semantics [4, 16]. However, there is no intention to fix the semantics within THF0. THF0 is designed to express problems syntactically, with the semantics being specified separately [3].

3 The THF0 Language

THF0 is a syntactically conservative extension of the untyped first-order TPTP language, adding the syntax for higher-order logic. Maintaining a consistent style between the first-order and higher-order languages facilitates easy adoption of the new language, through reuse or adaptation of existing infrastructure for processing TPTP format data, e.g., parsing tools, pretty-printing tools, system

testing, and result analysis, (see Section 5). A particular feature of the TPTP language, which has been maintained in THF0, is Prolog compatibility. This allows an annotated formula to be read with a single Prolog `read`/1 call, in the context of appropriate operator definitions. There are good reasons for maintaining Prolog compatibility [27].

Figure 1 presents an example problem encoded in THF0. The example is from the domain of basic set theory, stating the distributivity of union and intersection. It is a higher-order version of the first-order TPTP problem `SET171+3`, which employs first-order set theory to achieve a first-order encoding suitable for first-order theorem provers. The encoding in THF0 exploits the fact that higher-order logic provides a naturally built-in set theory, based on the idea of identifying sets with their characteristic functions. The higher-order encoding can be solved more efficiently than the first-order encoding [7, 8].

The first three annotated formulae of the example are *type declarations* that declare the type signatures of $\in$, $\cap$, and $\cup$. The `type` role is new, added to the TPTP language for THF0.[2] A simple type declaration has the form *constant_symbol*: *signature*. For example, the type declaration

```
thf(const_in,type,(
    in: ( $i > ( $i > $o ) > $o ) )).
```

declares the symbol `in` (for $\in$), to be of type $\iota \rightarrow (\iota \rightarrow o) \rightarrow o$. Thus `in` expects an element of type ι and a set of type $\iota \rightarrow o$ as its arguments. The mapping arrow is right associative. Type declarations use rules starting at `<thf_typed_const>` in the BNF. Note the use of the TPTP interpreted symbols `$i` and `$o` for the standard types ι and o, which are built in to THF0. In addition to `$i` and `$o`, THF0 has the defined type `$tType` that denotes the collection of all types, and `$iType`/`$oType` as synonyms for `$i`/`$o`. Further base types can be introduced (as `<system_type>`s) on the fly. For example, the following introduces the base type u together with a corresponding constant symbol `in_u` of type $u \rightarrow (u \rightarrow o) \rightarrow o$.

```
thf(type_u,type,(
    u: $tType)).

thf(const_in_u,type,(
    in_u: ( u > ( u > $o ) > $o ) )).
```

THF0 does not support polymorphism, product types or dependent types – such language constructs are addressed in THF1.

The next three annotated formulae of the example are *axioms* that specify the meanings of $\in$, $\cap$, and $\cup$. A THF0 logical formula can use the basic TPTP connectives, λ-abstraction using the `^` quantifier followed by a list of typed λ-bound variables in square brackets, and function application using the `@` connective. Additionally, universally and existentially quantified variables must be typed.

[2] It will also be used in the forthcoming typed first-order form (TFF) TPTP language.

```
%----------------------------------------------------------------------
%----Signatures for basic set theory predicates and functions.
thf(const_in,type,(
    in: $i > ( $i > $o ) > $o  )).

thf(const_intersection,type,(
    intersection: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) ) )).

thf(const_union,type,(
    union: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

%----Some axioms for basic set theory. These axioms define the set
%----operators as lambda-terms. The general idea is that sets are
%----represented by their characteristic functions.
thf(ax_in,axiom,(
    ( in
    = ( ^ [X: $i,S: ( $i > $o )] :
          ( S @ X ) ) ) )).

thf(ax_intersection,axiom,(
    ( intersection
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          & ( in @ U @ S2 ) ) ) ) )).

thf(ax_union,axiom,(
    ( union
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          | ( in @ U @ S2 ) ) ) ) )).

%----The distributivity of union over intersection.
thf(thm_distr,conjecture,(
    ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
      ( ( union @ A @ ( intersection @ B @ C ) )
      = ( intersection @ ( union @ A @ B ) @ ( union @ A @ C ) ) ) )).
%----------------------------------------------------------------------
```

Fig. 1. Distribution of Union over Intersection, encoded in THF0

For example, the axiom

```
thf(ax_in,axiom,(
    ( in
    = ( ^ [X: $i,S: ( $i > $o )] :
          ( S @ X ) ) ) )).
```

specifies the meaning of the constant $\in$ by equating it to $\lambda X_\iota.\lambda S_{\iota\to o}.(S\,X)$. Thus elementhood of an element X in a set S is reduced to applying the set

S (seen as its characteristic function S) to X. Using `in`, intersection is then equated to $\lambda S^1_{\iota\to o}.\lambda S^2_{\iota\to o}.\lambda U_\iota.(\in\ U\ S^1) \wedge (\in\ U\ S^2)$, and union is equated to $\lambda S^1_{\iota\to o}.\lambda S^2_{\iota\to o}.\lambda U_\iota.(\in\ U\ S^1) \vee (\in\ U\ S^2)$. Logical formulae use rules starting at `<thf_logic_formula>` in the BNF. The explicit function application operator `@` is necessary for parsing function application expressions in Prolog. Function application is left associative.

The last annotated formula of the example is the *conjecture* to be proved.

```
thf(thm_distr,conjecture,(
    ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
      ( ( union @ A @ ( intersection @ B @ C ) )
      = ( intersection @ ( union @ A @ B )
        @ ( union @ A @ C ) ) ) )).
```

This encodes $\forall A_{\iota\to o}.\forall B_{\iota\to o}.\forall C_{\iota\to o}.(A \cup (B \cap C)) = ((A \cup B) \cap (A \cup C))$. It uses rules starting at `<thf_quantified_formula>` in the BNF.

An advantage of higher-order logic over first-order logic is that the $\forall$ and $\exists$ quantifiers can be encoded using higher-order abstract syntax: Quantification is expressed using the logical constant symbols Π and Σ in connection with λ-abstraction. Higher-order systems typically make use of this feature in order to avoid introducing and supporting binders in addition to λ. THF0 provides the logical constants `!!` and `??` for Π and Σ respectively, and leaves use of the universal and existential quantifiers open to the user. Here is an encoding of the conjecture from the example, using `!!` instead of `!`.

```
thf(thm_distr,conjecture,(
    !! ( ^ [A: $i > $o,B: $i > $o,C: $i > $o] :
           ( ( union @ A @ ( intersection @ B @ C ) )
           = ( intersection @ ( union @ A @ B )
             @ ( union @ A @ C ) ) ) ) )).
```

Figure 2 presents the axioms from another example problem encoded in THF0. The example is from the domain of puzzles, and it encodes the following 'Knights and Knaves Puzzle':[3] *A very special island is inhabited only by knights and knaves. Knights always tell the truth. Knaves always lie. You meet two inhabitants: Zoey and Mel. Zoey tells you that Mel is a knave. Mel says, 'Neither Zoey nor I are knaves.' Can you determine who is a knight and who is a knave?* Puzzles of this kind have been discussed extensively in the AI literature. Here, we illustrate that an intuitive and straightforward encoding can be achieved in THF0. This encoding embeds formulas (terms of Boolean type) in terms and quantifies over variables of Boolean type; see for instance the `kk_6_2` axiom.

4 Type Checking THF0

The THF0 syntax provides a lot of flexibility. As a result, many syntactically correct expressions are meaningless because they are not well typed. For example, it does not make sense to say `& = ~` because the equated expressions have

[3] This puzzle was auto-generated by a computer program, written by Zac Ernst.

```
%----------------------------------------------------------------------
%----A very special island is inhabited only by knights and knaves.
thf(kk_6_1,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ islander )
     => ( ( is_a @ X @ knight )
        | ( is_a @ X @ knave ) ) ) )).

%----Knights always tell the truth.
thf(kk_6_2,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ knight )
     => ( ! [A: $o] :
            ( says @ X @ A )
       => A ) ) )).

%-----Knaves always lie.
thf(kk_6_3,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ knave )
     => ( ! [A: $o] : ( says @ X @ A )
       => ~ A ) ) )).

%----You meet two inhabitants: Zoey and Mel.
thf(kk_6_4,axiom,
    ( ( is_a @ zoey @ islander )
    & ( is_a @ mel @ islander ) )).

%----Zoey tells you that Mel is a knave.
thf(kk_6_5,axiom,
    ( says @ zoey @ ( is_a @ mel @ knave ) )).

%----Mel says, 'Neither Zoey nor I are knaves.'
thf(kk_6_6,axiom,
    ( says @ mel
    @ ~ ( ( is_a @ zoey @ knave )
        | ( is_a @ mel @ knave ) ) )).

%----Can you determine who is a knight and who is a knave?
thf(query,theorem,(
    ? [Y: $i,Z: $i] :
      ( ( Y = knight <~> Y = knave )
      & ( Z = knight <~> Z = knave )
      & ( is_a @ mel @ Y )
      & ( is_a @ zoey @ Z ) ) )).
%----------------------------------------------------------------------
```

Fig. 2. A 'Knights and Knaves Puzzle' encoded in THF0

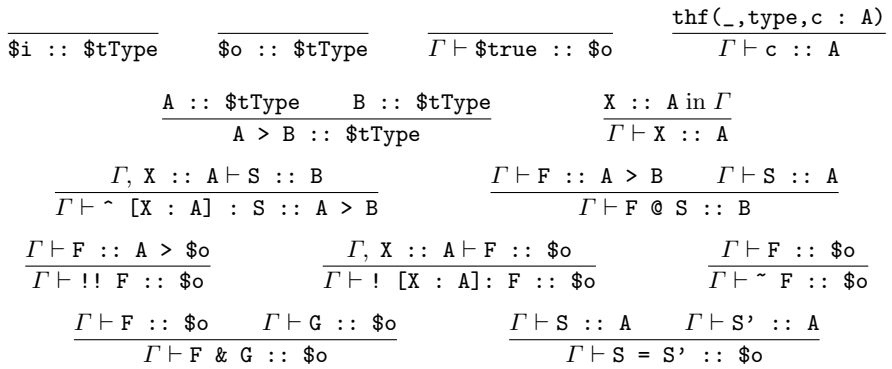

Fig. 3. Typing rules of THF0

different types. It is therefore necessary to type check expressions using an inference system. Representative typing rules are given in Fig. 3 (the missing rules are obviously analogous to those provided). Here `$tType` denotes the collection of all types, and `S :: A` denotes the judgement that `S` has type `A`.

The typing rules serve only to provide an intuition. The normative definition is given by representing THF0 in the logical framework LF [14]. LF is a dependent type theory related to Martin-Löf type theory [18], particularly suited to logic representations. In particular, the maintenance of the context, substitution, and α-conversion are handled by LF and do not have to be specified separately. A THF0 expression is well typed if its translation to LF is, and THF0 expressions can be type checked using existing tools such as Twelf [23]. In the following, Twelf syntax is used to describe the representation.

To represent THF0, a base signature Σ_0 is defined, as given in Figure 4. It is similar to the Harper et al. encoding of higher-order logic [14]. Every list L of TPTP formulas can be translated to a list of declarations Σ_L extending Σ_0. A list L is well-formed iff Σ_0, Σ_L is a well-formed Twelf signature. The signature of the translation is given in Figure 5.

In Figure 4, `$tType : type` means that the name `$tType` is introduced as an LF type. Its LF terms are the translations of THF0 types. The declaration `$tm : $tType -> type` introduces a symbol `$tm`, which has no analog in THF0: for every THF0 type `A` it declares an LF type `$tm A`, which holds the terms of type `A`. `$i` and `$o` are declared as base types, and `>` is the function type constructor. Here `->` is the LF symbol for function spaces, i.e., the declaration of `>` means that it takes two types as arguments and returns a type. Thus, on the ASCII level, the translation from THF0 types to LF is the identity. The symbols `^` through `??` declare term and formula constructors with their types. The binary connectives other than `&` have been omitted - they are declared just like `&`. In Twelf, equality and inequality are actually ternary symbols, because they take the type `A` of the equated terms as an additional argument. By using implicit arguments, Twelf is able to reconstruct `A` from the context, so that

equality behaves essentially like a binary symbol. Except for equality, the same ASCII notation can be used in Twelf as in THF0 so that on the ASCII level most cases of the translation are trivial. The translation of the binders `^`, `!`, and `?` is interesting: they are all expressed by the λ binder of Twelf. For example, if F is translated to F', then $\forall x_{\$i}.F$ is translated to $!(\lambda x_{\$tm\ \$i}.F')$. Since the Twelf ASCII syntax for $\lambda x_{\$tm\ \$i}.F'$ is simply `[x:$tm $i] F'`, on the ASCII level the translation of binders consists of simply inserting `$tm` and dropping a colon. For all binders, the type `A` of the bound variable is an implicit argument and thus reconstructed by Twelf. Of course, most of the term constructors can be defined in terms of only a few primitive ones. In particular, the Twelf symbol `!` can be defined as the composition of `!!` and `^`, and similarly `?`. However, since they are irrelevant for type checking, definitions are not used here.

```
$tType  : type.
$tm     : $tType -> type.
$i      : $tType.
$o      : $tType.
>       : $tType -> $tType -> $tType.

^       : ($tm A -> $tm B) -> $tm (A > B).
@       : $tm(A > B) -> $tm A -> $tm B.
$true   : $tm $o.
$false  : $tm $o.
~       : $tm $o -> $tm $o.
&       : $tm $o -> $tm $o -> $tm $o.
==      : $tm A -> $tm A -> $tm $o.
!=      : $tm A -> $tm A -> $tm $o.

!       : ($tm A -> $tm $o) -> $tm $o.
?       : ($tm A -> $tm $o) -> $tm $o.
!!      : ($tm(A > $o)) -> $tm $o.
??      : ($tm(A > $o)) -> $tm $o.

$istrue : $tm $o -> type.
```

Fig. 4. The base signature for Twelf

THF0	LF
types	terms of type `$tType`
terms of type `A`	terms of type `$tm A`
formulas	terms of type `$tm $o`

Fig. 5. Correspondences between THF0 and the LF representation

Figure 6 shows the translation of the formulae from Figure 1. A THF0 type declaration for a constant `c` with type `A` is translated to the Twelf declaration `c : $tm A`. An axiom or conjecture `F` is translated to a Twelf declaration for the type `$istrue F'`, where `F'` is the translation of `F`. In the latter case the Twelf name of the declaration is irrelevant for type-checking. By using the role, i.e., axiom or conjecture, as the Twelf name, the role of the original TPTP formula can be recovered from its Twelf translation.

```
in           : $tm($i > ($i > $o) > $o).
intersection : $tm(($i > $o) > ($i > $o) > ($i > $o)).
union        : $tm(($i > $o) > ($i > $o) > ($i > $o)).
axiom        : $istrue in == ^[X : $tm $i] ^[S : $tm($i > $o)](S @ X).
axiom        : $istrue intersection ==
                    ^[S1: $tm($i > o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
                     ((in @ U @ S1) & (in @ U @ S2)).
axiom        : $istrue union ==
                    ^[S1: $tm($i > $o)] ^[S2: $tm($i > $o)] ^[U: $tm $i]
                     (in @ U @ S1) | (in @ U @ S2).
conjecture   : $istrue
               !![A: $tm($i > $o)] !![B: $tm($i > $o)] !![C: $tm($i > $o)]
                  ((union @ A @ (intersection @ B @ C))
                  == (intersection @ (union @ A @ B) @ (union @ A @ C))).
```

Fig. 6. Twelf translation of Figure 1

Via the Curry-Howard correspondence [10, 17], the representation of THF0 in Twelf can be extended to represent proofs – it is necessary only to add declarations for the proof rules to Σ_0, i.e., β-η-conversion, and the rules for the connectives and quantifiers. If future versions of THF0 provide a standard format for explicit proof terms, Twelf can serve as a neutral trusted proof checker.

5 TPTP Resources

The first-order TPTP provides a range of resources to support use of the problem library and related infrastructure. Many of these resources are immediately applicable to the higher-order setting, while some require changes to reflect the new features in the THF0 language.

The main resource for users is the TPTP problem library itself. At the time of writing around 100 THF problems have been collected, and are being prepared for addition to a THF extension of the library. THF file names have an `@` separator between the abstract problem name and the version number (corresponding to the `-` in CNF problem names and the `+` in FOF problem names), e.g., the example problem in Figure 1 will be put in `SET171@4.p`. The existing header fields of TPTP problems have been slightly extended to deal with higher-order features. First, the `Status` field, which records the semantic status of the

problem in terms of the SZS ontology, provide status values for each semantics of interest [3, 4]. Second, the `Syntax` field, which records statistics of syntactic characteristics of the problem, has been extended to include counts of the new connectives that are part of THF0.

Tools that support the TPTP include the tptp2X and tptp4X utilities, which read, analyse, transform, and output TPTP problems. tptp2X is written in Prolog, and it has been extended to read, analyse, and output problems written in THF0. The translation to Twelf syntax has been implemented as a tptp2X format module, producing files that can be type-checked directly. The Twelf translation has been used to type-check (and in some cases correct) the THF problems collected thus far. The extended tptp2X is part of TPTP v3.4.0. tptp4X is based on the `JJParser` library of code written in C. This will be extended to cope with higher-order formulae.

The flipside of the TPTP problem library is the TSTP solution library. Once the higher-order part of the TPTP problem library is in place it is planned to extend the TSTP to include results from higher-order ATP systems on the THF problems. The harnesses used for building the first-order TSTP will be used as-is for the higher-order extension.

A first competition "happening" for higher-order ATP systems that can read THF0 will be held at IJCAR 2008. This event will be similar to the CASC competition for first-order ATP systems, but with a less formal evaluation phase. It will exploit and test the THF0 language.

6 Conclusion

This paper has described, with examples, the core of the TPTP language for higher-order logic (Church's simple type theory) – THF0. The TPTP infrastructure is being extended to support problems, solutions, tools, and evaluation of ATP systems using the THF0 language. Development and adoption of the THF0 language and associated TPTP infrastructure will support research and development in automated higher-order reasoning, providing leverage for progress leading to effective and successful application. To date only the LEO II system [8] is known to use the THF0 language. It is hoped that more high-order reasoning systems and tools will adopt the THF language, making easy and direct communication between the systems and tools possible.

Acknowledgements: Chad Brown contributed to the design of the THF language. Allen Van Gelder contributed to the development of the THF syntax and BNF. Frank Theiss and Arnaud Fietzke implemented the LEO-II parser for THF0 language and provided useful feedback.

References

1. P.B. Andrews, M. Bishop, S. Issar, Nesmith. D., F. Pfenning, and H. Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.

2. M. Beeson. Otter-lambda, a Theorem-prover with Untyped Lambda-unification. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.
3. C. Benzmüller and C. Brown. A Structured Set of Higher-Order Problems. In J. Hurd and T. Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, number 3606 in Lecture Notes in Artificial Intelligence, pages 66–81. Springer-Verlag, 2005.
4. C. Benzmüller, C. Brown, and M. Kohlhase. Higher-order Semantics and Extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.
5. C. Benzmüller and M. Kohlhase. LEO - A Higher-Order Theorem Prover. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 139–143. Springer-Verlag, 1998.
6. C. Benzmüller and L. Paulson. Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. In C. Benzmüller, C. Brown, J. Siekmann, and R. Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*. IfCoLog, 2007. To appear.
7. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined Reasoning by Automated Cooperation. *Journal of Applied Logic*, 2008. In print.
8. C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2008. This proceedings.
9. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:5668, 1940.
10. H.B. Curry and R. Feys. *Combinatory Logic I*. North Holland, Amsterdam, 1958.
11. F. Frege. *Grundgesetze der Arithmetik*. Jena, 1893,1903.
12. P. Godefroid. Software Model Checking: the VeriSoft Approach. Technical Report Technical Memorandum ITD-03-44189G, Bell Labs, Lisle, USA, 2003.
13. M. Gordon and T. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
14. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.
15. J. Harrison. HOL Light: A Tutorial Introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design*, number 1166 in Lecture Notes in Computer Science, pages 265–269. Springer-Verlag, 1996.
16. L. Henkin. Completeness in the Theory of Types. *Journal of Symbolic Logic*, 15:81–91, 1950.
17. W. Howard. The Formulas-as-types Notion of Construction. In J. Seldin and J. Hindley, editors, *To H B Curry, Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.
18. P. Martin-Löf. An Intuitionistic Theory of Types. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, pages 127–172. Oxford University Press, 1973.
19. C. Matuszek, Cabral J., M. Witbrock, and J. DeOliveira. An Introduction to the Syntax and Content of Cyc. In Baral C., editor, *Proceedings of the 2006 AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its*

Applications to Knowledge Representation and Question Answering, pages 44–49, 2006.
20. I. Niles and A. Pease. Towards A Standard Upper Ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*, pages 2–9, 2001.
21. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer-Verlag, 2002.
22. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer-Verlag, 1996.
23. F. Pfenning and C. Schürmann. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in Lecture Notes in Artificial Intelligence, pages 202–206. Springer-Verlag, 1999.
24. P. Rudnicki. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332, 1992.
25. J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with omega. *Journal of Applied Logic*, 4(4):533–559, 2006.
26. G. Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Computer Science Symposium in Russia*, number 4649 in Lecture Notes in Computer Science, pages 7–23. Springer-Verlag, 2007.
27. G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.
28. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.
29. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
30. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
31. A. Van Gelder and G. Sutcliffe. Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 156–161. Springer-Verlag, 2006.
32. E. Zermelo. Über Grenzzahlen und Mengenbereiche. *Fundamenta Mathematicae*, 16:29–47, 1930.

A BNF for THF0

The BNF uses a modified BNF meta-language that separates syntactic, semantic, lexical, and character-macro rules [27]. Syntactic rules use the standard `::=` separator, semantic constraints on the syntactic rules use a `:==` separator, rules that produce tokens from the lexical level use a `::-` separator, and the bottom level character-macros are defined by regular expressions in rules using a `:::` separator. The BNF is easy to translate into parser-generator (`lex/yacc`, `antlr`, etc.) input [31].

The syntactic and semantic grammar rules for THF0 are presented here. The rules defining tokens and character macros are available from the TPTP web site, www.tptp.org.

```
%----------------------------------------------------------------------------
%----Files. Empty file is OK.
<TPTP_file>          ::= <TPTP_input>*
<TPTP_input>         ::= <annotated_formula> | <include>

%----Formula records
<annotated_formula>  ::= <thf_annotated>
<thf_annotated>      ::= thf(<name>,<formula_role>,<thf_formula><annotations>).
<annotations>        ::= <null> | ,<source><optional_info>
%----In derivations the annotated formulae names must be unique, so that
%----parent references (see <inference_record>) are unambiguous.

%----Types for problems.
<formula_role>       ::= <lower_word>
<formula_role>       :== axiom | hypothesis | definition | assumption |
                         lemma | theorem | conjecture | negated_conjecture |
                         plain | fi_domain | fi_functors | fi_predicates |
                         type | unknown
%----------------------------------------------------------------------------
%----THF0 formulae. All formulae must be closed.
<thf_formula>        ::= <thf_logic_formula> | <thf_typed_const>
<thf_logic_formula>  ::= <thf_binary_formula> | <thf_unitary_formula>
<thf_binary_formula> ::= <thf_pair_binary> | <thf_tuple_binary>
%----Only some binary connectives can be written without ()s.
%----There's no precedence among binary connectives
<thf_pair_binary>    ::= <thf_unitary_formula> <thf_pair_connective>
                         <thf_unitary_formula>
%----Associative connectives & and | are in <assoc_formula>.
<thf_tuple_binary>   ::= <thf_or_formula> | <thf_and_formula> |
                         <thf_apply_formula>
<thf_or_formula>     ::= <thf_unitary_formula> <vline> <thf_unitary_formula> |
                         <thf_or_formula> <vline> <thf_unitary_formula>
<thf_and_formula>    ::= <thf_unitary_formula> & <thf_unitary_formula> |
                         <thf_and_formula> & <thf_unitary_formula>
<thf_apply_formula>  ::= <thf_unitary_formula> @ <thf_unitary_formula> |
                         <thf_apply_formula> @ <thf_unitary_formula>
%----<thf_unitary_formula> are in ()s or do not have a <binary_connective>
%----at the top level. Essentially, a <thf_unitary_formula> is any lambda
%----expression that "has enough parentheses" to be used inside a larger
%----lambda expression. However, lambda notation might not be used.
<thf_unitary_formula> ::= <thf_quantified_formula> | <thf_abstraction> |
                          <thf_unary_formula> | <thf_atom> |
                          (<thf_logic_formula>)
<thf_quantified_formula> ::= <thf_quantified_var> | <thf_quantified_novar>
<thf_quantified_var> ::= <quantifier> [<thf_variable_list>] :
                         <thf_unitary_formula>
<thf_quantified_novar>   ::= <thf_quantifier> (<thf_unitary_formula>)
%----@ (denoting apply) is left-associative and lambda is right-associative.
<thf_abstraction>    ::= <thf_lambda> [<thf_variable_list>] :
                         <thf_unitary_formula>
```

```
<thf_variable_list>  ::= <thf_variable> | <thf_variable>,<thf_variable_list>
<thf_variable>       ::= <variable> | <thf_typed_variable>
<thf_typed_variable> ::= <variable> : <thf_top_level_type>
%----Unary connectives bind more tightly than binary. The negated formula
%----must be ()ed because a ~ is also a term.
<thf_unary_formula>  ::= <thf_unary_connective> (<thf_logic_formula>)

%----An <thf_typed_const> is a global assertion that the atom is in this type.
<thf_typed_const>    ::= <constant> : <thf_top_level_type> | (<thf_typed_const>)

%----THF atoms
<thf_atom>           ::= <constant> | <defined_constant> | <system_constant> |
                         <variable> | <thf_conn_term>
%----<defined_constant> is really a <defined_prop>, but that's the syntax rule
%----used. <thf_atom> can also be <defined_type>, but they are syntactically
%----captured by <defined_constant>. Ditto for <system_*>.

%----<thf_top_level_type> appears after ":", where a type is being specified
%----for a term or variable.
<thf_unitary_type>   ::= <constant> | <variable> | <defined_type> |
                         <system_type> | (<thf_binary_type>)
<thf_top_level_type> ::= <constant> | <variable> | <defined_type> |
                         <system_type> | <thf_binary_type>
<thf_binary_type>    ::= <thf_mapping_type> | (<thf_binary_type>)
<thf_mapping_type>   ::= <thf_unitary_type> <arrow> <thf_unitary_type> |
                         <thf_unitary_type> <arrow> <thf_mapping_type>
%--------------------------------------------------------------------------
%----Special higher order terms
<thf_conn_term>      ::= <thf_quantifier> | <thf_pair_connective> |
                         <assoc_connective> | <thf_unary_connective>

%----Connectives - THF
<thf_lambda>         ::= ^
<thf_quantifier>     ::= !! | ??
<thf_pair_connective> ::= <defined_infix_pred> | <binary_connective>
<thf_unary_connective> ::= <unary_connective>
%----Connectives - FOF
<quantifier>         ::= ! | ?
<binary_connective>  ::= <=> | => | <= | <~> | ~<vline> | ~&
<assoc_connective>   ::= <vline> | &
<unary_connective>   ::= ~

%----Types for THF and TFF
<defined_type>       ::= <atomic_defined_word>
<defined_type>       :== $oType | $o | $iType | $i | $tType
%----$oType/$o is the Boolean type, i.e., the type of $true and $false.
%----$iType/$i is type of individuals. $tType is the type of all types.
<system_type>        ::= <atomic_system_word>

%----First order atoms
<defined_prop>       :== <atomic_defined_word>
<defined_prop>       :== $true | $false
<defined_infix_pred> ::= = | !=

%----First order terms
<constant>           ::= <functor>
<functor>            ::= <atomic_word>
<defined_constant>   ::= <atomic_defined_word>
<system_constant>    ::= <atomic_system_word>
<variable>           ::= <upper_word>
%--------------------------------------------------------------------------
%----General purpose
<name>               ::= <atomic_word> | <unsigned_integer>
<atomic_word>        ::= <lower_word> | <single_quoted>
<atomic_defined_word> ::= <dollar_word>
<atomic_system_word> ::= <dollar_dollar_word>
<null>               ::=
%--------------------------------------------------------------------------
```