TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism

Cezary Kaliszyk
University of Innsbruck, Austria

Geoff Sutcliffe
University of Miami, USA

Florian Rabe
Jacobs University Bremen, Germany

Abstract

The TPTP world is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP language is one of the keys to the success of the TPTP world. Originally the TPTP world supported only first-order clause normal form (CNF). Over the years support for full first-order form (FOF), monomorphic typed first-order form (TF0), rank-1 polymorphic typed first-order form (TF1), and monomorphic typed higher-order form (TH0) have been added. The TF0, TF1, and TH0 languages also include constructs for arithmetic. This paper introduces the TH1 form, an extension of TH0 with TF1-style rank-1 polymorphism. TH1 is designed to be easy to process by existing reasoning tools that support ML-style polymorphism. The hope is that TH1 will be implemented in many popular ATP systems for typed higher-order logic.

1 Introduction

The TPTP world [18] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP world includes the TPTP problem library, the TSTP solution library, the TMTP model library, standards for writing ATP problems and reporting ATP solutions, tools and services for processing ATP problems and solutions, and it supports the CADE ATP System Competition (CASC). Various parts of the TPTP world have been deployed in a range of applications, in both academia and industry. The web page `http://www.tptp.org` provides access to all components.

The TPTP language is one of the keys to the success of the TPTP world. The language is used for writing both TPTP problems and TSTP solutions, which enables convenient communication between different systems and researchers. Originally the TPTP world supported only first-order clause normal form (CNF) [21]. Over the years support for full first-order form (FOF) [17], monomorphic typed first-order form (TF0) [20], rank-1 polymorphic typed first-order form (TF1) [3], and monomorphic typed higher-order form (TH0) [19] have been added. The TF0, TF1, and TH0 languages also include constructs for arithmetic. These extensions provide orthogonal features that are natural to combine. In fact, several users have already used ad-hoc definitions of combined languages, and [9] already suggested a systematic way to combine specifications.

This paper introduces the TH1 format, a combination of the features of TH0 with TF1-style rank-1 polymorphism. TH1 is designed to be easy to process by existing reasoning tools that support ML-style polymorphism.

In: P. Fontaine, S. Schulz, J. Urban (eds.): Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning (PAAR 2016), Coimbra, Portugal, 02-07-2016, published at http://ceur-ws.org

It opens the door to useful middleware, such as monomorphizers and other translation tools that encode polymorphism in FOF or TH0. Many existing ATP and ITP systems for higher-order logic already implement some kind of polymorphism. TH1 has been designed in interaction with those systems' developers, and can provide an interoperability layer for their systems. The hope is that TH1 will be implemented in many popular ATP systems for typed higher-order logic.

Figure 1 shows the relationships between the various TPTP language forms, leading to TH1. TH1 combines two existing orthogonal extensions to the basic FOF language: the polymorphic TF1 and the higher-order TH0.[1] Therefore, this paper first presents the preceding TF0, TF1, and TH0 forms, whose features are then inherited in the TH1 form.

The remainder of this paper is organized as follows: Section 2 introduces the generic TPTP language, with a brief review of the well-known first-order form. Sections 2.1 and 2.2 describe the monomorphic and polymorphic typed first-order forms, and Section 2.3 describes the monomorphic typed higher-order form. Sections 3 and 4 are the heart of the paper, describing the syntax and semantics of the polymorphic higher-order form respectively. Section 5 explains the LF-based type checking system for the typed languages. Section 6 concludes.

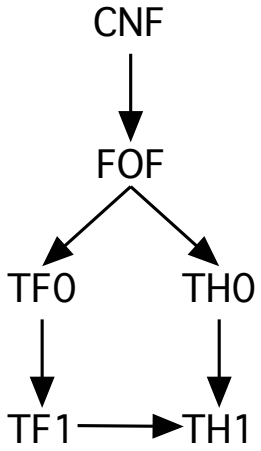

Figure 1: The DAG of TPTP Language Forms

2 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 top level building blocks of the TPTP language are *annotated formulae*. An annotated formula has the form:

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

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

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

[1] Note that while TH1 adopts features of TF1, TH0 is independent of TF0.

2.1 The Monomorphic Typed First-order Form TF0

TF0 extends the basic FOF language with *types* and *type declarations*. Every function and predicate symbol is declared before its use, with a *type signature* that specifies the types of the symbol's arguments and result.

TF0 types τ have the following forms:

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

User-defined types are declared (before their use) to be of the kind `$tType`, in annotated formulae with a `type` role – see Figure 2 for examples. All symbols share the same namespace; in particular, a type cannot have the same name as a function or predicate symbol.

TF0 type signatures ς have the following forms:

- individual types τ;
- $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_i are the argument types, and $\tilde{\tau}$ is the result type. Argument types and the result type for a function cannot be `$o`, and the result type for a predicate must be `$o`. If $n = 1$ the parentheses are omitted.

The type signatures of uninterpreted symbols are declared like types, in annotated formulae with a `type` role – see Figure 2 for examples. The type of `=` is ad hoc polymorphic over all types except `$o`, with both arguments having the same type and the result type being `$o`. The types of arithmetic predicates and functions are ad hoc polymorphic over the arithmetic types; see [20] for details.

In logical formulae the arguments to functions and predicates must respect the symbol's type signature. Variables must be typed in their quantification, and used accordingly. For example, in Figure 2 the predicate `chased` is declared to have the type signature

```
(dog * cat) > $o
```

so an example of a well-formed formula is

```
! [X: dog] : chased(X,garfield)
```

because `X` is of type `dog` and `garfield` is of type `cat`.

Figure 2 illustrates some TF0 formulae, whose conjecture can be proved from the axioms (it is the TPTP problem `PUZ130_1.p`).

2.2 The Polymorphic Typed First-order Form TF1

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

TF1 types τ have the following forms:

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

Type constructors are declared (before their use) to be of the kind `($tType * ··· * $tType) > $tType`, in annotated formulae with a `type` role. The type constructor's arity is the number of "`$tType`"s before the `>`. If the arity is zero the `>` is omitted, and the type constructor is a monomorphic user-defined type constant as in TF0, else it is polymorphic. If the arity is less than two the parentheses are omitted. In the example of Figure 3, `cup_of` is a unary type constructor that is is used to construct the type `cup_of(beverage))`.

TF1 type signatures ς have the following forms:

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

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

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

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

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

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

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

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

Figure 2: TF0 Formulae

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

The binder `!>` in the last form denotes universal quantification in the style of $\lambda\Pi$ calculi. It is only used at the top level in polymorphic type signatures. All type variables must be of type `$tType`; more complex type variables, e.g., `$tType > $tType` are beyond rank-1 polymorphism. In the example of Figure 3, the type signature for `mixture` is polymorphic. As in TF0, arithmetic symbols and equality are ad hoc polymorphic.

Type variables that are bound by `!>` without occurring in a type signature's body, e.g., as in the last example, are called *phantom type variables*. These make it possible to specify operations and relations directly on types and provide a convenient way to encode type classes. For example, a polymorphic propositional constant `is_linear` can be declared with the type signature `!>[A : $tType]: $o`, and used as a guard to restrict the axioms specifying that a binary predicate `less_eq` with the type signature `!>[A: $tType]: ((A * A) > $o)` is a linear order to those types that satisfy the `is_linear` predicate, i.e., a formula of the form `! [A: $tType] : ( (is_linear @ A) => (`*axiom about less_eq*`))`.

In logical formulae the application of symbols must respect the symbol's type signature. Additionally, every use of a polymorphic symbol (except ad hoc polymorphic arithmetic and equality symbols) must explicitly specify the type instance. A function or predicate symbol with a type signature `!>[`α_1`: $tType, ... ,`α_m`: $tType]: ((`τ_1 `*` $\cdots$ `>` τ_n `>` $\tilde{\tau}$`)` must be applied to m type arguments and n term arguments. For example, in Figure 3 below, the type contructor `cup_of`, the function `full_cup`, the function `mixture`, and the predicate `help_stay_awake`, are declared to have the type signatures

```
$tType > $tType
beverage > cup_of(beverage)
!>[BeverageOrSyrup: $tType] : ((BeverageOrSyrup * syrup) > BeverageOrSyrup)
cup_of(beverage) > $o
```

so an example of a well-formed formula is

```
! [S: syrup] : help_stay_awake(full_cup(mixture(beverage,coffee,S)))
```

because `beverage` provides the one type argument for `mixture`, `coffee` is of that type, `S` is of type `syrup`, and the result type is `beverage` as required by `full_cup`, whose whose result type is in turn `cup_of(beverage)` as required by `help_stay_awake`. As TF1 is rank-1 polymorphic, the type arguments must be actual types; in particular `$tType` and `!>`-binders cannot occur in type arguments of polymorphic symbols.

Figure 3 illustrates some TF1 formulae, whose conjecture can be proved from the axioms (it is a variant of the TPTP problem `PUZ139_1.p`).

2.3 The Monomorphic Typed Higher-order Form TH0

TH0 extends FOF with higher-order notions, including adoption of curried form for type declarations, lambda terms (with a lambda binder), symbol application, and two other new binders.

TH0 types τ have the following forms:

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

TH0 type signatures ς have the following forms:

- individual types τ.

```
%----------------------------------------------------------------------------
ff(beverage_type,type,( beverage: $tType )).
tff(syrup_type,type,( syrup: $tType )).
tff(cup_of_type,type,( cup_of: $tType > $tType )).
tff(full_cup_type,type,( full_cup: beverage > cup_of(beverage) )).
tff(coffee_type,type,( coffee: beverage )).
tff(vanilla_type,type,( vanilla: syrup )).
tff(caramel_type,type,( caramel: syrup )).
tff(help_stay_awake_type,type,( help_stay_awake: cup_of(beverage) > $o )).

tff(mixture_type,type,(
    mixture: !>[BeverageOrSyrup: $tType] :
               ( ( BeverageOrSyrup * syrup ) > BeverageOrSyrup ) )).

%----Coffee keeps you awake
tff(mixture_of_coffee_help_stay_awake,axiom,(
    ! [S: syrup] : help_stay_awake(full_cup(mixture(beverage,coffee,S))) )).

%----Coffee mixed with a syrup or two helps you stay awake
tff(syrup_coffee_help_stay_awake,conjecture,(
    help_stay_awake(full_cup(
      mixture(beverage,coffee,mixture(syrup,caramel,vanilla)))) )).
%----------------------------------------------------------------------------
```

Figure 3: TF1 Formulae

The $(\tau_1 > \cdots > \tau_n > \tilde{\tau})$ form is promoted to be a type in TH0 (and TH1) forms, so that variables in formulae can be quantified as function types. The curried form means that TH0 provides the possibility of partial application. As in TF0, arithmetic symbols and equality are ad hoc polymorphic.

The new binary connective `@` represents application (explicit use of `@` is required – symbols cannot simply be juxtaposed) – see Figure 4 for examples. There are three new binders: `^`, `@+`, and `@-`, for λ, ϵ (indefinite description, aka choice), and ι (definite description). Use of all the connectives as terms is also supported – this is quite straightforward because connectives are equivalent (in Henkin semantics - see below) to corresponding lambda abstractions. The arithmetic predicates and functions, and the equality operator, can also be used as terms, but must be applied to a term of a known type (which must be an arithmetic type for the arithmetic symbols). This formula illustrates the use of conjunction as a term, in a definition of $false$,

```
thf(and_false,definition,(
    ( ( & @ $false )
    = ( ^ [Q: $o] : $false ) ) )).
```

In logical formulae the application of symbols must respect the symbol's type signature. For example, in Figure 4 the symbols `mix` and `hot` are declared to have the type signatures

```
beverage > syrup > beverage
beverage > $o
```

so an example of a well-formed formula is

```
! [S: syrup] : (hot @ (mix @ coffee @ S))
```

because `S` is of type `syrup`, `coffee` is of type `beverage`, and `mix` the result type is `beverage` as required by `hot`. Partial application results in a lambda term. For example, `mix @ coffee` produces a lambda term of type `syrup > beverage`.

Figure 4 illustrates some TH0 formulae, whose conjecture can be proved from the axioms (it is a variant of the TPTP problem `PUZ140^1.p`). The following formula illustrates the definite description binder (the indefinite and definite description binders are not used in Figure 4) for a set whose elements of type `$i` are all equal,

```
thf(one_set,axiom,
    ( ! [Z: $i] : ( ( p @ Z ) => ( Z = y ) )
   => ( ( @-[X: $i] : ( p @ X ) ) = y ) )).
```

The semantics for TH0 is Henkin semantics with choice (Henkin semantics by definition also includes Boolean and functional extensionality) [1, 8].

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

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

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

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

Figure 4: TH0 Formulae

3 The TH1 Syntax

TH1 combines the higher-order features of TH0 (curried form, lambda terms, description, partial application) with the polymorphic features of TF1 (type constructors, type variables, polymorphic symbols). TH1 also adds five new polymorphic constants.

TH1 types τ have the following forms:

- the predefined types `$i` and `$o`;
- user-defined n-ary type constructors applied to n type arguments;
- type variables (which must be quantified by `!>`);
- `(`τ_1 `>` $\cdots$ `>` τ_n `>` $\tilde{\tau}$`)` for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type.

TH1 type signatures ς have the following forms:

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

In logical formulae the application of symbols must respect the symbol's type signature. Additionally, as in TF1, every use of a polymorphic symbol must explicitly specify the type instance by providing type arguments. However, in TH1 the application to terms may be partial, i.e., a symbol with a type signature

`!>[`α_1`: $tType, ... ,`α_m`: $tType]: (`τ_1 `>` $\cdots$ `>` τ_n `>` $\tilde{\tau}$`)`

must be applied to m type arguments and *up to* n term arguments. For example, given the type declarations in Figure 5 below, some examples of well-formed formulae are

```
idempotent @ bird @ ^ [X: bird] : X
! [M: map @ bird @ $o] : ( bird_lookup @ bird @ $o @ M @ tweety )
! [T: $tType] : ( idempotent @ T @ ^ [X: T] : X ).
```

TH1 has five new polymorphic constants: `!!` for Π (universal quantification), `??` for Σ (existential quantification), `@@+` for ϵ (indefinite description, aka choice), `@@-` for ι (definite description), and `@=` (equality).[2] Each of these must be instantiated by applying them to exactly one type argument, e.g.,

`? [B: bird] : ( (@= @ bird) @ tweety @ B ).`

Figure 5 illustrates some TH1 formulae, whose conjecture can be proved from the axioms. It declares and axiomatizes `lookup` and `update` operations on maps, then conjectures that `update` is idempotent for certain keys and values. Figure 6 illustrates the `!!` universal quantification operator by defining `apply_both` as the function that supplies its argument to `the_function` as both its arguments, then conjecturing that the two are equal for all types. Figure 7 illustrates the `@@-` indefinite description operator by asserting that the function `has_fixed_point` has a fixed point (some `X` such that `f(X) = X`), and then conjecturing that applying the function to some fixed point simply returns that fixed point. In informal mathematics, there is no guarantee that the two "some fixed point" expressions are equal, e.g., if `has_fixed_point` is the identity function then all points are fixed points. However, in logic, "some fixed point" has a fixed interpretation in a given model, so the conjecture is a theorem. Figure 8 illustrates the `@=` equality connective by defining the higher-order `is_symmetric` predicate that takes a predicate as an argument and returns true iff the argument is a symmetric predicate, and then conjecturing that `@=` is symmetric (which it is).

```
%----------------------------------------------------------------------------
thf(bird_type,type,( bird: $tType )).
thf(tweety_type,type,( tweety: bird )).

thf(list_type,type,( list: $tType > $tType )).
thf(map_type,type,( map: $tType > $tType > $tType )).

tff(bird_lookup_type,type,(
    bird_lookup: !>[A: $tType,B: $tType] : ( ( map @ A @ B ) > A > B ) )).
thf(bird_update_type,type,(
    bird_update: !>[A: $tType,B: $tType] :
                   ( ( map @ A @ B ) > A > B > ( map @ A @ B ) ) )).
thf(idempotent_type,type,( idempotent: ( !>[A: $tType] : ( A > A ) > $o ) )).

thf(bird_lookup_update_same,axiom,(
    ! [RangeType: $tType,Map: ( map @ bird @ RangeType ),
       Key: bird,Value: RangeType] :
      ( ( bird_lookup @ bird @ RangeType @
          ( bird_update @ bird @ RangeType @ Map @ Key @ Value ) @ Key )
      = Value ) )).

thf(idempotent_def,definition,(
    ! [A: $tType,F: ( A > A )] :
      ( ( idempotent @ A @ F )
      = ( ! [X: A] : ( ( F @ ( F @ X ) ) = ( F @ X ) ) ) ) )).

thf(higher_order_conjecture,conjecture,(
    ! [Value: ( list @ $i )] :
      ( idempotent @ ( map @ bird @ ( list @ $i ) )
      @ ^ [Map: ( map @ bird @ ( list @ $i ) )] :
          ( bird_update @ bird @ ( list @ $i ) @ Map @ tweety @ Value ) ) )).
%----------------------------------------------------------------------------
```

Figure 5: TH1 Formulae

4 Semantics

The semantics of TH1 have to extend that of TH0 and that of TF1, to maximize the uniformity of the TPTP. Additionally, the semantics have be compatible with that of existing higher-order logic proof assistants and

[2] `!!` and `??` used to be in TH0, but they have been moved out to be in only TH1 now.

```
%------------------------------------------------------------------------------
thf(a_type,type,( a_type: $tType )).
thf(apply_both_type,type,( apply_both: a_type > a_type )).
thf(the_function_type,type,( the_function: a_type > a_type > a_type )).

thf(apply_both_defn,axiom,(
    apply_both = ( ^ [X: a_type] : ( the_function @ X @ X ) ) )).

thf(prove_this,conjecture,
    ( !! @ a_type
    @ ^ [Y: a_type] : ( ( the_function @ Y @ Y ) = ( apply_both @ Y ) ) )).
%------------------------------------------------------------------------------
```

Figure 6: TH1 `!!` Example

```
%------------------------------------------------------------------------------
thf(a_type,type,( a_type: $tType )).
thf(has_fixed_point_type,type,( has_fixed_point: a_type > a_type )).

thf(apply_has_fixed_point,axiom,(
    ? [X: a_type] : ( ( has_fixed_point @ X ) = X ) )).

thf(conj,conjecture,
    ( ( has_fixed_point
      @ ( @@+ @ a_type
        @ ^ [Y: a_type] : ( ( has_fixed_point @ Y ) = Y ) ) )
    = ( @@+ @ a_type
      @ ^ [Y: a_type] : ( ( has_fixed_point @ Y ) = Y ) ) )).
%------------------------------------------------------------------------------
```

Figure 7: TH1 `@@-` Example

```
%------------------------------------------------------------------------------
thf(a_type,type,(
    a_type: $tType )).

thf(is_symmetric_type,type,(
    is_symmetric: ( ( $i > a_type ) > ( $i > a_type ) > $o ) > $o )).

thf(is_symmetric_def,axiom,(
    ! [Predicate: ( $i > a_type ) > ( $i > a_type ) > $o] :
      ( ( is_symmetric @ Predicate )
      = ( ! [X: $i > a_type,Y: $i > a_type] :
            ( ( Predicate @ X @ Y ) = ( Predicate @ Y @ X ) ) ) ) )).

thf(conj,conjecture,
    ( is_symmetric @ ( @= @ ( $i > a_type ) ) )).
%------------------------------------------------------------------------------
```

Figure 8: TH1 @= Example

countermodel finders, so as to provide a problem exchange language for such tools.

The semantics of TH1 is, as for TH0, the Henkin semantics with choice, extended to shallow polymorphism [5]. Assuming the syntax of types and terms introduced in the previous sections, the semantics is given as a set of inference rules that define provability in TH1. The provability relation is the same as the provability induced by major HOL-based proof assistants. The inference rules are closest to those of HOL Light [7], because those rules are most concise, but are the same as those of other systems, including Isabelle/HOL [13] and HOL4 [15], as well as frameworks that allow higher-order proofs to be specified independently of the proof assistant, e.g., OpenTheory [10].

The first three rules introduce the reflexivity and transitivity of equality, and assumption:

$$\frac{\texttt{t} : \texttt{bool}}{\vdash \texttt{t} = \texttt{t}} \qquad \frac{\Gamma_1 \vdash \texttt{s} = \texttt{t} \quad \Gamma_2 \vdash \texttt{t} = \texttt{u}}{\Gamma_1 \cup \Gamma_2 \vdash \texttt{s} = \texttt{u}} \qquad \frac{}{\{\texttt{p}\} \vdash \texttt{p}}$$

The next two define application and abstraction over equality theorems:

$$\frac{\Gamma_1 \vdash \texttt{f} = \texttt{g} \quad \Gamma_2 \vdash \texttt{x} = \texttt{y} \quad \texttt{f} : \alpha \texttt{ -> } \beta \quad \texttt{x} : \alpha}{\Gamma_1 \cup \Gamma_2 \vdash \texttt{f @ x = g @ y}} \qquad \frac{\Gamma_1 \vdash \texttt{s} = \texttt{t}}{\Gamma_1 \vdash \texttt{(\^{}[x:}\alpha\texttt{]:s) = (\^{}[x:}\alpha\texttt{]:t)}}$$

The next three correspond to beta reduction, *modus ponens* for equalities, and deduction of logical equivalence from deductions in both directions:

$$\frac{\texttt{t} : \alpha \texttt{ -> } \beta}{\vdash \texttt{(\^{}[x:}\alpha\texttt{]:t)(x:}\alpha\texttt{) = t}} \qquad \frac{\Gamma_1 \vdash \texttt{s} = \texttt{t} \quad \Gamma_2 \vdash \texttt{s}}{\Gamma_1 \cup \Gamma_2 \vdash \texttt{t}} \qquad \frac{\Gamma_1 \vdash \texttt{p} \quad \Gamma_2 \vdash \texttt{q}}{\Gamma_1 - \{\texttt{q}\} \cup \Gamma_2 - \{\texttt{p}\} \vdash \texttt{p = q}}$$

The final two rules allow for instantiation of types and terms in proven theorems:

$$\frac{\Gamma[\alpha_1, ..., \alpha_n] \vdash \texttt{p}[\alpha_1, ..., \alpha_n]}{\Gamma[\gamma_1, ..., \gamma_n] \vdash \texttt{p}[\gamma_1, ..., \gamma_n]} \qquad \frac{\Gamma[\texttt{x}_1, ..., \texttt{x}_n] \vdash \texttt{p}[\texttt{x}_1, ..., \texttt{x}_n]}{\Gamma[\texttt{t}_1, ..., \texttt{t}_n] \vdash \texttt{p}[\texttt{t}_1, ..., \texttt{t}_n]}$$

In order to achieve the exact semantics of the higher-order logic provers, it is necessary to add axioms that correspond to those of the provers. For example, HOL Light introduces three axioms: eta (from which classical logic follows), infinity (which allows the definition an infinite type of individuals, out of which the natural numbers are carved), and choice. To achieve the Henkin semantics (including the induced classical logic) the eta axiom is added. The infinity axiom is not needed because TH1 problems introduce the necessary types as assumptions. Two instances of the axiom of choice are given: one for choice and one for description. They specify the operators `@++` and `@--`. None of the axioms need any inference assumptions, so they can be given in the TPTP syntax.

```
thf(eta,axiom,(
    ! [A: $tType,B: $tType,A0: ( A > B )] :
      ( ( ^ [A1: A] : ( A0 @ A1 ) ) = A0 ) )).

thf(choice_type,type,(
    @@+: !>[A: $tType] : ( ( A > $o ) > A ) )).

thf(choice,axiom,(
    ! [A: $tType,P: ( A > $o ),A0: A] :
      ( ( P @ A0 ) => ( P @ ( @@+ @ A @ P ) ) ) )).

thf(description_type,type,(
    @@-: !>[A: $tType] : ( ( A > $o ) > A ) )).

thf(description,axiom,(
    ! [A: $tType,P: ( A > $o ),A0: A] :
      ( ( P @ A0 )
     => ! [B: A] :
          ( ( ( P @ B ) => ( A = B ) )
         => ( P @ ( @@- @ A @ P ) ) ) ) )).
```

5 Type Checking

Well-typed TPTP formulae are defined uniformly for CNF, FOF, TF0, TF1, TH0, and TH1 using a set of typing rules. We use the logical framework LF [6] to give these rules. The use of LF is not critical and other representations of the typing rules would work similarly. But LF has the advantage of being extremely concise, providing the structural rules (including the closure under substitution) for free, and — via the Twelf tool for LF [14] — providing an implementation of the type checker out of the box.

The type-checking process is split into two steps. The first step translates a TPTP problem P to a Twelf theory $\underline{P}$. This step is implemented as part of the TPTP2X tool, and is a relatively simple induction on the context-free grammar for TPTP. The second step runs Twelf to check the theory $\underline{TH1}, \underline{P}$. This includes, in particular, the context-sensitive parts of the language. This is the same process as used for TH0 [2]. The steps are described in more detail below. The Twelf theory TH1 is given in Appendix A.

Translation to Twelf

Besides bridging the minor syntactic differences between TPTP and Twelf, this step performs two basic structural checks:

- It reject formulae not allowed in the respective language form, e.g., it rejects polymorphic type signatures in TF0 formula.
- It translates all formulae into TH1.

The translation into TH1 proceeds as follows:

- CNF to FOF: This well-known and straightforward translation has been part of the TPTP tool suite for some time.
- FOF to TF0: This is essentially an inclusion. The only subtlety is that the untyped FOF constants and quantifiers must become typed. This is done by assuming that every untyped n-ary function or predicate symbol has type signature `($i * ... * $i) > $i` or `($i * ... * $i) > $o`, respectively. Moreover, all FOF variables are taken to range over the type `$i`.
- TF0 to TF1, and TH0 to TH1: These translations are inclusions.
- TF0 to TH0, and TF1 to TH1: These translations are almost inclusions. The only subtlety is that TF0/1 type signatures using `*` are translated to TH0/1 type signatures using `>`.

Type-Checking in LF

The inference system is a straightforward variant of the one used for TH0 [2]. The **rules for types** are:

```
$tType: type
$i    : $tType
$o    : $tType
>     : $tType -> $tType -> $tType     (infix)
```

These declare the LF type `$tType` of TPTP types, as well as `$i`, `$o`, and $>$ as constructors for it. User-declared n-ary type operators `T` are translated to Twelf constants `T: $tType -> ... -> $tType -> $tType`.

The **rules for terms** are

```
$tm   : $tType -> type
^     : ($tm A -> $tm B) -> $tm (A > B)
@     : $tm (A > B) -> $tm A -> $tm B   (infix)
```

These declare the LF type `$tm A` of TPTP terms of type `A` for every TPTP type `A:$tType`, as well as constructors for `^` (using higher-order abstract syntax) and `@`. The latter two constructors take two implicit arguments `A` and `B` of type `$tType`, which are inferred by Twelf. User-declared (monomorphic) constants `f` of type τ are translated to Twelf constants `f: $tm` $\underline{\tau}$, where $\underline{\tau}$ is the recursive translation of τ.

The **rules for the remaining term constructors** are as follows (omitting the constructors for propositional formulae described in [2]):

```
!     : ($tm A -> $tm $o) -> $tm $o
?     : ($tm A -> $tm $o) -> $tm $o
@+    : ($tm A -> $tm $o) -> $tm A
@-    : ($tm A -> $tm $o) -> $tm A
==    :  $tm A -> $tm A   -> $tm $o
```

For example, `!` takes an LF function from a TPTP type `A` to a TPTP formula as an argument, and returns a TPTP formula. Because the LF function $\lambda X : \texttt{\$tm A.t}$ is written `[X:$tm $A] t` in Twelf's concrete syntax, the TPTP formula `![X:A]:F` becomes `![X:$tm A]F` in Twelf syntax.

There is no need for **rules for proofs**, except for a truth predicate

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

This allows user-declared axioms and conjectures `a` asserting `F` to be represented as Twelf constants `a:$istrue F`.

Rank-1 **polymorphism** comes for free when using LF [9]. In fact, the type checker for TH0 has always allowed polymorphism, which is suppressed for TH0 in the context-free grammar. The reason is that LF's dependent function space can be used to represent type signatures that quantify over type variables. First, note that the dependent function type $\Pi\alpha\texttt{:\$tType.T}$ of LF is written $\{\alpha\texttt{:\$tType}\}$ `T` in Twelf. A user-declared polymorphic constant `f` that takes type arguments $\alpha_1, \ldots, \alpha_n$ and has type τ is represented as the Twelf constant $\texttt{f:}\{\alpha_1\}\ldots\{\alpha_n\}$ `$tm` $\underline{\tau}$. User-declared polymorphic axioms and conjectures are represented accordingly. Finally, any occurrence of `f` with type arguments $\tau_1, \ldots, \tau_n$ is translated to the Twelf term $f\ \underline{\tau_1}\ \ldots\ \underline{\tau_n}$.

In addition to the five term constructors, whose types are given above, TH1 also introduces five corresponding constants. Besides being a part of TH1, they serve as examples of how polymorphic constants are represented in Twelf:

```
!! : {A: $tType} $tm ((A > $o) > $o)
?? : {A: $tType} $tm ((A > $o) > $o)
@@+: {A: $tType} $tm ((A > $o) > A )
@@-: {A: $tType} $tm ((A > $o) > A )
@= : {A: $tType} $tm ( A > A   > $o)
```

For example, the TPTP formula `!! @ A @ F` is represented in Twelf as `(!! A) @ F`.

6 Conclusion

This paper has described the polymorphic typed-higher order form (TH1) in the TPTP language family, which extends the monomorphic TH0 language with TF1-style rank-1 polymorphism. A type-checker for the language has been developed. Now that the language has been specified, TH1 problems can be added to the TPTP problem library. Already hundreds of problems have been collected from various sources, e.g., the HOL(y)Hammer and Sledgehammer exports from HOL [11, 16] and [12]. The addition of TH1 problems to the TPTP should provide the impetus for ATP system developers to extend their systems to TH1. In fact many systems could process TH1 with modifications to their parsers only, including LEO-III [22] and Why3 [4]. The hope is that TH1 will be implemented in many popular ATP systems for typed higher-order logic. Beyond that, a TH1 division could be added to CASC to further stimulate development of robust TH1 ATP systems that are useful and easily deployed in applications, leading to increased and highly effective use.

Acknowledgements.

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

References

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

[2] C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 - The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, number 5195 in Lecture Notes in Artificial Intelligence, pages 491–506. Springer-Verlag, 2008.

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

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *STTT*, 17(6):709–727, 2015.

[5] M. Gordon and A. Pitts. The HOL Logic. In M. Gordon and T. Melham, editors, *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*, pages 191–232. Cambridge University Press, 1993.

[6] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.

[7] J. Harrison. HOL Light: An Overview. In T. Nipkow and C. Urban, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, number 5674 in Lecture Notes in Computer Science, pages 60–66. Springer-Verlag, 2009.

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

[9] F. Horozal and F. Rabe. Formal Logic Definitions for Interchange Languages. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Proceedings of the International Conference on Intelligent Computer Mathematics*, number 9150 in Lecture Notes in Computer Science, pages 171–186. Springer-Verlag, 2015.

[10] J. Hurd. The OpenTheory Standard Theory Library. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the 3rd International Symposium on NASA Formal Methods*, number 6617 in Lecture Notes in Computer Science, pages 177–191, 2011.

[11] Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

[12] T. Matsuzaki, H. Iwane, M. Kobayashi, Y. Zhan, R. Fukasaku, J. Kudo, H. Anai, and N. Arai. Race against the Teens - Benchmarking Mechanized Math on Pre-university Problems. In N. Olivetti and A. Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page To appear, 2016.

[13] 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.

[14] 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.

[15] K. Slind and M. Norrish. A Brief Overview of HOL4. In O. Mohamed, C. Munoz, and S. Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, number 5170 in Lecture Notes in Computer Science, pages 28–32. Springer-Verlag, 2008.

[16] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

[17] 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.

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

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

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

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

[22] M. Wisniewski, A. Steen, and C. Benzmüller. The Leo-III Project. In A. Bolotov and M. Kerber, editors, *Proceedings of the Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.

A The Twelf TH1 Theory

```
$kind : type.

%% types (i, o, functions and products)
$tf   : $kind -> type.
$type : $kind.
$tType = $tf $type.
$i : $tType.
$o : $tType.
> : $tType -> $tType -> $tType.                    %infix right 10 >.
* : $tType -> $tType -> $tType.                    %infix right 10 *.

%% terms (lambda and application)
$tm : $tType -> type.                             %prefix 2 $tm.
^ : ($tm A -> $tm B) -> $tm (A > B).              %prefix 10 ^.
@ : $tm (A > B) -> $tm A -> $tm B.                %infix left 10 @.

%% connectives
$true : $tm $o.
& : $tm $o -> $tm $o -> $tm $o.                  %infix left 5 &.
| : $tm $o -> $tm $o -> $tm $o.                  %infix left 5 |.
~ : $tm $o -> $tm $o.

$false : $tm $o
    = ~ $true.
=>  : $tm $o -> $tm $o -> $tm $o
    = [a][b](a | ~ b).                            %infix none 5 =>.
<=  : $tm $o -> $tm $o -> $tm $o
    = [a][b](b => a).                             %infix none 5 <=.
<=> : $tm $o -> $tm $o -> $tm $o
    = [a][b]((a => b) & (b => a)).                %infix none 5 <=>.
<~> : $tm $o -> $tm $o -> $tm $o
    = [a][b] ~ (a <=> b).                         %infix none 5 <~>.
~&  : $tm $o -> $tm $o -> $tm $o
    = [a][b] ~ (a & b).                           %infix none 5 ~&.
~|  : $tm $o -> $tm $o -> $tm $o
    = [a][b] ~ (a | b).                           %infix none 5 ~|.

%% equality over a type A
== : $tm A -> $tm A -> $tm $o.                    %infix none 10 ==.
!= : $tm A -> $tm A -> $tm $o
   = [x][y] ~ (x == y).                           %infix none 10 !=.

%% quantification over a type A, taking a meta-level function
! : ($tm A -> $tm $o) -> $tm $o.                  %prefix 5 !.
? : ($tm A -> $tm $o) -> $tm $o
  = [f] ~ (! [x] ~ (f x)).                        %prefix 5 ?.

%% description and choice (as binders)
@+ : ($tm A -> $tm $o) -> $tm A.                 %prefix 5 @+.
@- : ($tm A -> $tm $o) -> $tm A.                 %prefix 5 @-.
```

```
%% polymorphic constants for the 4 binders and equality; type argument is mandatoy
!! : {A} $tm (A > $o) > $o
   = [A] ^ [f] ! [x] (f @ x).                    %prefix 5 !!.
?? : {A} $tm (A > $o) > $o
   = [A] ^ [f] ? [x] (f @ x).                    %prefix 5 !!.
@@+: {A} $tm (A > $o) > A
   = [A] ^ [f] @+ [x] (f @ x).                   %prefix 5 @@+.
@@-: {A} $tm (A > $o) > A
   = [A] ^ [f] @- [x] (f @ x).                   %prefix 5 @@-.
@= : {A} $tm A > A > $o
   = [A] ^ [x] ^ [y] x == y.                     %prefix 5 @=.

%% equality as a polymorphic constant with type inference (no type argument)
$tptp_equal : $tm A > A > $o
   = @= A.                                       %prefix 5 @=.

%% equality and quantifiers over types
=t= : $tm A -> $tm A -> $tm $o.                  %infix none 10 =t=.
!t! : ($tType -> $tm $o) -> $tm $o.              %prefix 5 !t!.
?t? : ($tType -> $tm $o) -> $tm $o.              %prefix 5 ?t?.

%% dependent sum and product type
%% ?* : ($tm A -> $tType) -> $tType.               %prefix 5 ?*.
%% !> : ($tm A -> $tType) -> $tType.               %prefix 5 !>.

%% arithmetic types as a subtype
$arithdom   : type.
$_int       : $arithdom.
$_rat       : $arithdom.
$_real      : $arithdom.
$arithtype  : $arithdom -> $tType.
$int        : $tType = $arithtype $_int.
$rat        : $tType = $arithtype $_rat.
$real       : $tType = $arithtype $_real.

%% dummy constants to represent arithmetic literals
$intliteral  : $tm $int.
$ratliteral  : $tm $rat.
$realliteral : $tm $real.

%% arithmetic operations in higher-order style for an arbitrary arithmetic type
$less      : $tm ($arithtype D) > ($arithtype D) > $o.
$lesseq    : $tm ($arithtype D) > ($arithtype D) > $o.
$greater   : $tm ($arithtype D) > ($arithtype D) > $o.
$greatereq : $tm ($arithtype D) > ($arithtype D) > $o.

$uminus    : $tm ($arithtype D) > ($arithtype D).
$sum       : $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$difference: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$product   : $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$quotient  : $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$quotient_e: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$quotient_t: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$quotient_f: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$remainder_e: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$remainder_t: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).
$remainder_f: $tm ($arithtype D) > ($arithtype D) > ($arithtype D).

$is_int    : $tm ($arithtype D) > $o.
$is_rat    : $tm ($arithtype D) > $o.
$is_real   : $tm ($arithtype D) > $o.

$to_int    : $tm ($arithtype D) > $int.
$to_rat    : $tm ($arithtype D) > $rat.
$to_real   : $tm ($arithtype D) > $real.

%% truth predicate
$istrue : $tm $o -> type.                        %prefix 1 $istrue.
```