

TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism

Jasmin Christian Blanchette¹ and Andrei Paskevich^{2,3}

¹ Fakultät für Informatik, Technische Universität München, Germany

² LRI, Université Paris-Sud 11, CNRS, France

³ ProVal, INRIA Saclay-Île-de-France, France

Abstract. The TPTP World is a well-established infrastructure for automatic theorem provers. It defines several concrete syntaxes, notably an untyped first-order form (FOF) and a typed first-order form (TFF0), that have become de facto standards in the automated reasoning community. This paper introduces the TFF1 format, an extension of TFF0 with rank-1 polymorphism. It presents its syntax, typing rules, and semantics, as well as a sound and complete translation to TFF0. The format is designed to be easy to process by existing reasoning tools that support ML-style polymorphism. It opens the door to useful middleware, such as monomorphizers and other translation tools that encode polymorphism in FOF or TFF0. Ultimately, the hope is that TFF1 will be implemented in popular automatic theorem provers.

1 Introduction

The TPTP World [27] is a well-established infrastructure for supporting research, development, and deployment of automated reasoning tools. It owes its name to its vast problem library, the Thousands of Problems for Theorem Provers (TPTP) [26]. In addition, it specifies concrete syntaxes for problems and solutions: Dozens of reasoning tools implement the TPTP untyped clause normal form (CNF) and first-order form (FOF) for classical first-order logic with equality.

It has often been argued that the gap between the features supported by provers and those needed by applications is too wide, and that rich interchange formats are needed to address this disconnect [15, 17, 22, 24, 31]. A growing number of reasoners can process the recently introduced TPTP “core” typed first-order form (TFF0) [29], with monomorphic types and interpreted arithmetic [14, 21], or the corresponding higher-order form (THF0) [2]; a polymorphic version of THF0, the full THF, is in the works [28].

Despite the variety of this offering, there is a strong desire in part of the automated reasoning community for a portable *polymorphic first-order format*. Many applications require polymorphism, notably interactive theorem provers and program specification languages; but lacking a suitable syntax, applications and provers must communicate via monomorphic formats. To make matters worse, there is no entirely satisfactory way to eliminate polymorphism: Monomorphization algorithms are necessarily incomplete [7, §2], and it is difficult to encode polymorphism in a complete yet also sound and efficient manner, especially in the presence of interpreted types. Tool authors are

reduced to developing their own monomorphizers and type encodings, often using sub-optimal schemes. Polymorphism arguably belongs in provers, where it can be implemented simply and efficiently, as demonstrated by the SMT (satisfiability modulo theories) solver Alt-Ergo [6].

This paper introduces the TFF1 format, an extension of TFF0 with rank-1 polymorphism. The extension was designed with the participation of members of the TPTP community, reflecting its needs. Besides compatibility with TFF0 and conceptual integrity with the upcoming full THF, an important design goal was to ensure that the format can easily be processed by existing reasoning tools that support ML-style polymorphism. TFF1 also opens the door to useful middleware, such as monomorphizers and other tools that encode polymorphism in FOF or TFF0.

For SMT solvers, the SMT-LIB 2 format [1] specifies a classical many-sorted logic with equality and interpreted arithmetic, much in the style of TFF0 but with parametric symbol declarations (overloading). Polymorphism would make sense there as well, as witnessed by Alt-Ergo. However, the SMT community is still recovering from the upgrade to SMT-LIB 2 and busy defining a standard proof format [4]; implementers would not welcome yet another feature at this point. Moreover, with its support for arithmetic, TFF1 is a reasonable format to implement in an SMT solver if polymorphism is desired.

This paper is structured as follows. Section 2 specifies the TFF1 syntax, and Section 3 specifies its typing rules and semantics. Section 4 presents a translation from TFF1 to TFF0. Section 5 bridges the gap with ML-style formalisms by defining two preprocessing steps that address type quantifiers in TFF1 type signatures and formulas. Section 6 briefly reviews the applications that already implement TFF1. Section 7 considers related work in the TPTP and SMT communities: existing polymorphic formalisms and translation schemes to untyped or monomorphic logics.

This specification extends, rather than replaces, the TFF0 specification [29]. The parts that TFF1 inherits directly from TFF0, such as the concrete syntax for the logical connectives and the (optional) arithmetic constructs, are described in more detail there.

2 Syntax

Briefly, the types, terms, and formulas of TFF1 are analogous to those of TFF0, except that function and predicate symbols can be declared to be polymorphic, types can contain type variables, and n -ary type constructors are allowed. Type variables in type signatures and formulas are explicitly bound. Instances of polymorphic symbols are specified by explicit type arguments, rather than inferred.

Types. The *types* of TFF1 are built from *type variables* and *type constructors* of fixed arities. Nullary type constructors are called *type constants*. The usual conventions of TPTP apply: Type variables start with an uppercase letter and type constructors with a lowercase letter. The types `A`, `list(A)`, `list(bird)`, and `map(nat, list(B))` are all examples of well-formed types. A type is *polymorphic* if it contains any type variables; otherwise, it is *monomorphic*.

As in TFF0, the type `$i` of individuals is predefined but has no fixed semantics, whereas the arithmetic types `$int`, `$rat`, and `$real` are modeled by \mathbb{Z} , \mathbb{Q} , and \mathbb{R} . The

TFF0 specification [29] defines the semantics of the arithmetic types and their operations. It is perfectly acceptable for a TFF implementation to restrict itself to “pure TFFk,” without arithmetic. TFFk with arithmetic is sometimes labeled “TFAk.”

Type Signatures. Each function and predicate symbol occurring in a formula must be associated with a *type signature* that specifies the types of the arguments and, for functions, the return type. Type signatures can take any of the following forms:

- (a) a type (predefined or user-defined);
- (b) the Boolean pseudotype $\$0$;
- (c) $(\tau_1 * \dots * \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_1, \dots, τ_n are types and $\tilde{\tau}$ is a type or $\$0$;
- (d) $!>[\alpha_1 : \$tType, \dots, \alpha_n : \$tType] : \zeta$ for $n > 0$, where $\alpha_1, \dots, \alpha_n$ are distinct type variables and ζ has one of the previous three forms.

The parentheses in form (c) are omitted if $n = 1$. The binder $!>$ in form (d) denotes universal quantification. If ζ is of form (c), it must be enclosed in parentheses. All type variables must be bound by a $!>$ -binder.

Here are a few examples: (a) $\$int$, $monkey$, $banana$; (b) $\$0$; (c) $monkey > banana$, $(monkey * banana) > \$0$; (d) $!>[A : \$tType] : ((A * list(A)) > list(A))$.

Form (a) is used for monomorphic constants; form (b), for propositional constants, including the predefined symbols $\$true$ and $\$false$; form (c), for monomorphic functions and predicates; and form (d), for polymorphic functions and predicates. It is often convenient to regard all forms above as instances of the general syntax

$$!>[\alpha_1 : \$tType, \dots, \alpha_m : \$tType] : ((\tau_1 * \dots * \tau_n) > \tilde{\tau})$$

where m and n can be 0.

Type variables that are bound by $!>$ without occurring in the type signature’s body 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, we can declare a polymorphic propositional constant `is_linear` with the type signature $!>[A : \$tType] : \0 and use it as a guard to restrict the axioms specifying that a binary predicate `less_eq` with the type signature $!>[A : \$tType] : ((A * A) > \$0)$ is a linear order to those types that satisfy the `is_linear` predicate.

Type Declarations. Type constructors can optionally be declared. The following declarations introduce a type constant `bird`, a unary type constructor `list`, and a binary type constructor `map`:

```
tff(bird_t, type, bird: $tType).
tff(list_t, type, list: $tType > $tType).
tff(map_t, type, map: ($tType * $tType) > $tType).
```

If a type constructor is used before being declared, its arity is determined by the first occurrence. Any later declaration must give it the same arity.

A declaration of a function or predicate symbol specifies its type signature. Every type variable occurring in a type signature must be bound by a $!>$ -binder. The following declarations introduce a monomorphic constant `pi`, a polymorphic predicate `is_empty`, and a pair of polymorphic functions `cons` and `lookup`:

```

tff(pi_t, type, pi: $real).
tff(is_empty_t, type, is_empty : !>[A : $tType]: (list(A) > $o)).
tff(cons_t, type,
    cons : !>[A : $tType]: ((A * list(A)) > list(A))).
tff(lookup_t, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).

```

If a function or predicate symbol is used before being declared, a default type signature is assumed: $(\$_i * \dots * \$_i) > \$_i$ for functions and $(\$_i * \dots * \$_i) > \$_o$ for predicates. If a symbol is declared after its first use, the declared signature must agree with the assumed signature. If a type constructor, function symbol, or predicate symbol is declared more than once, it must be given the same type signature up to renaming of bound type variables. All symbols share the same namespace; in particular, a type constructor cannot have the same name as a function or predicate symbol.

Function and Predicate Application. To keep the required type inference to a minimum, every use of a polymorphic symbol must explicitly specify the type instance. A function or predicate symbol with a type signature

$$!>[\alpha_1 : \$tType, \dots, \alpha_m : \$tType]: ((\tau_1 * \dots * \tau_n) > \tilde{\tau})$$

must be applied to m type arguments and n term arguments. Given the above type signatures for `is_empty`, `cons`, and `lookup`, the term `lookup($int, list(A), M, 2)` and the atom `is_empty($i, cons($i, X, nil($i)))` are well-formed and contain free occurrences of the type variable `A` and the term variables `M` and `X`.

In keeping with TFF1's rank-1 polymorphic nature, type variables can only be instantiated with actual types. In particular, `$o`, `$tType`, and `!>`-binders cannot occur in type arguments of polymorphic symbols.

For systems that implement type inference, the following nonstandard extension of TFF1 might be useful. When a type argument of a polymorphic symbol can be inferred automatically, it may be replaced with the wildcard `$_`. For example:

```
is_empty($_, cons($_, X, nil($_)))
```

Although `nil`'s type argument cannot be inferred locally from the types of its term arguments (since there are none), the Hindley–Milner type inference can deduce it. The producer of a TFF1 problem must be aware of the type inference algorithm implemented in the consumer to omit only redundant type arguments.

Type and Term Variables. Every variable in a TFF1 formula must be bound. The variable's type must be specified at binding time:

```

tff(bird_list_not_empty, axiom,
    ![B : bird, Bs : list(bird)]:
    ~ is_empty(bird, cons(bird, B, Bs))).

```

If the type and the preceding colon (`:`) are omitted, the variable is given type `$i`. Every type variable occurring in a TFF1 formula (whether in a type argument or in the type of a bound variable) must also be bound, with the pseudotype `$tType`:

```
tff(lookup_update_same, axiom,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
    lookup(A, B, update(A, B, M, K, V), K) = V).
```

A single quantifier cluster can bind both type variables and term variables. Universal and existential quantifiers over type variables are allowed under the propositional connectives, including equivalence, as well as under other quantifiers over type variables, but not in the scope of a quantifier over a term variable. Rationale: A statement of the form “for every integer k , there exists a type α such that ...” effectively makes α a dependent type. On such statements, type skolemization (Section 5.2) is impossible, and there is no easy translation to ML-style polymorphic formalisms. Moreover, type handling in an automatic prover would be more difficult were such constructions allowed, since they require paramodulation into types.

On the other hand, all the notions and procedures described in this specification—except for type skolemization—are independent of this restriction. The rules of type checking and the notion of interpretation are directly applicable to unrestricted formulas. The encoding into a monomorphic logic (Section 4) is sound and complete on unrestricted formulas, and the proofs require no adjustments. This prepares the ground for TFF2, which is expected to lift the restriction and support more elaborate forms of dependent types. Implementations of TFF1 are encouraged to support unrestricted formulas, treating them according to the semantics given here, if practicable.

Terms and Formulas. Apart from the differences described above, the terms and formulas of TFF1 are identical to those of TFF0, as defined in the TFF0 specification [29].

Example. The following problem gives the general flavor of TFF1. It first declares and axiomatizes lookup and update operations on maps, then conjectures that update is idempotent for fixed keys and values. Its SZS status [25] is Theorem.

```
tff(map_t, type, map : ($tType * $tType) > $tType).
tff(lookup_t, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).
tff(update_t, type,
    update : !>[A : $tType, B : $tType]:
        ((map(A, B) * A * B) > map(A, B))).

tff(lookup_update_same, axiom,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
    lookup(A, B, update(A, B, M, K, V), K) = V).

tff(lookup_update_diff, axiom,
    ![A : $tType, B : $tType, M : map(A, B), V : B, K : A, L : A]:
    (K != L => lookup(A, B, update(A, B, M, K, V), L) =
        lookup(A, B, M, L))).

tff(map_ext, axiom,
    ![A : $tType, B : $tType, M : map(A, B), N : map(A, B)]:
    ((![K : A]: lookup(A, B, M, K) = lookup(A, B, N, K)) =>
        M = N)).
```

```

tff(update_idem, conjecture,
  ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
  update(A, B, update(A, B, M, K, V), K, V) =
  update(A, B, M, K, V)).

```

3 Type Checking and Semantics

Notation. Starting with this section, we use standard mathematical notation to write types, terms, and formulas. Our conventions for metavariables are summarized below.

Type variables:	α, β	Function symbols:	f, g
Type constructors:	κ	Predicate symbols:	p, q
Types:	σ, τ	Terms:	s, t
Term variables:	u, v	Formulas:	φ, ψ

Possibly empty lists of types and terms are denoted by $\bar{\sigma}$ and \bar{t} , respectively.

We use the symbols \times , \rightarrow , and \forall to write type signatures, and write \circ (lowercase omicron) for the Boolean pseudotype $\$o$. It is convenient to treat equality (\approx), negation (\neg), conjunction (\wedge), and universal quantification (\forall) as logical symbols and regard disequality ($\not\approx$), disjunction (\vee), implication (\rightarrow), reverse implication (\leftarrow), equivalence (\leftrightarrow), inequivalence ($\not\leftrightarrow$), and existential quantification (\exists) as abbreviations. Equality could be seen as a polymorphic predicate with the type signature $\forall\alpha. \alpha \times \alpha \rightarrow \circ$, but the type instance is implicitly specified by the type of either argument, instead of explicitly via a type argument; hence, it is preferable to treat it as a logical symbol.

The set of type variables occurring freely in a formula φ (in the type arguments of polymorphic symbols or in the types of bound variables) is denoted by $FV_{\top}(\varphi)$. The set of free term variables of φ is denoted by $FV(\varphi)$. The formula φ is *closed* if both $FV_{\top}(\varphi)$ and $FV(\varphi)$ are empty.

A *type substitution* ρ is a mapping of type variables to types. A *monomorphic* type substitution maps every type variable either to itself or to a monomorphic type. Given a function h , the expression $h[x \mapsto a]$ denotes the function that maps x to a and every other element y in h 's domain to $h(y)$.

Type Checking. Let γ be a *type context*, a function that maps every variable symbol to a type. A type judgment $\gamma \vdash t : \tau$ expresses that the term t is *well-typed* and has type τ in context γ . A type judgment $\gamma \vdash \varphi : \circ$ expresses that the formula φ is *well-typed* in γ . We write $f : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$ and $p : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \circ$ to specify type signatures of function and predicate symbols, where m and n can be 0.

The typing rules of TFF1 are given below:

$$\begin{array}{c}
\overline{\gamma \vdash u : \gamma(u)} \\
\\
\frac{f : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau \quad \gamma \vdash t_1 : \tau_1 \rho \quad \dots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash f(\alpha_1 \rho, \dots, \alpha_m \rho, t_1, \dots, t_n) : \tau \rho} \\
\\
\frac{p : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \circ \quad \gamma \vdash t_1 : \tau_1 \rho \quad \dots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash p(\alpha_1 \rho, \dots, \alpha_m \rho, t_1, \dots, t_n) : \circ}
\end{array}$$

$$\begin{array}{c}
\frac{\gamma \vdash s : \tau \quad \gamma \vdash t : \tau}{\gamma \vdash s \approx t : \mathbf{o}} \qquad \frac{\gamma \vdash \varphi : \mathbf{o} \quad \gamma \vdash \psi : \mathbf{o}}{\gamma \vdash \varphi \wedge \psi : \mathbf{o}} \\
\\
\frac{\gamma \vdash \varphi : \mathbf{o}}{\gamma \vdash \neg \varphi : \mathbf{o}} \qquad \frac{\gamma[u \mapsto \tau] \vdash \varphi : \mathbf{o}}{\gamma \vdash \forall u : \tau. \varphi : \mathbf{o}} \qquad \frac{\gamma \vdash \varphi[\alpha'/\alpha] : \mathbf{o}}{\gamma \vdash \forall \alpha. \varphi : \mathbf{o}}
\end{array}$$

In the last rule, α' is an arbitrary type variable that occurs neither in φ nor in the values of γ . The renaming is necessary to reject formulas such as $\forall \alpha. \forall u : \alpha. \forall v : \alpha. u \approx v$, where the types of u and v are actually different. To simplify the subsequent definitions, we assume from now on that no type variable can be both free and bound in the same formula; we call this the *no-clash assumption*. As a result, we can avoid explicit renaming of type variables, and the last typing rule's premise becomes $\gamma \vdash \varphi : \mathbf{o}$.

Since every type variable in a polymorphic type signature must be bound in it, it is impossible for a term to have two different types in the same context.

A closed TFF1 formula φ is *well-typed* iff the judgment $\gamma \vdash \varphi : \mathbf{o}$ is derivable for any γ . Obviously, if a closed formula is well-typed in one type context, it is well-typed in any other one; hence, we omit γ and write $\vdash \varphi : \mathbf{o}$. Closed well-typed formulas are called *sentences*.

Semantics. An interpretation \mathfrak{J} for a given set of type constructors, function symbols, and predicate symbols is constructed as follows. First, we fix a nonempty collection \mathbb{D} of nonempty sets, the *domains*. The union of all domains is called the *universe*, \mathbb{U} .

An n -ary type constructor κ is interpreted as a function $\kappa^{\mathfrak{J}} : \mathbb{D}^n \rightarrow \mathbb{D}$. Let θ be a *type valuation*, a function that maps every type variable to a domain. Types are evaluated according to the following equations:

$$\llbracket \alpha \rrbracket_{\theta}^{\mathfrak{J}} \triangleq \theta(\alpha) \qquad \llbracket \kappa(\tau_1, \dots, \tau_n) \rrbracket_{\theta}^{\mathfrak{J}} \triangleq \kappa^{\mathfrak{J}}(\llbracket \tau_1 \rrbracket_{\theta}^{\mathfrak{J}}, \dots, \llbracket \tau_n \rrbracket_{\theta}^{\mathfrak{J}})$$

Since type evaluation depends only on the values of θ on the type variables occurring in a type, we write $\llbracket \tau \rrbracket^{\mathfrak{J}}$ to denote the domain of a monomorphic type τ . We use the notation $[\alpha_1 \mapsto D_1, \dots, \alpha_m \mapsto D_m]$ to specify θ for types whose free type variables are among $\alpha_1, \dots, \alpha_m$.

A predicate symbol $p : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \mathbf{o}$ is interpreted as a relation $p^{\mathfrak{J}} \subseteq \mathbb{D}^m \times \mathbb{U}^n$. A function symbol $f : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$ is interpreted as a function $f^{\mathfrak{J}}$ on $\mathbb{D}^m \times \mathbb{U}^n$ that maps any m domains D_1, \dots, D_m and n universe elements to an element of $\llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}$, where θ maps each α_i to D_i .

Let ξ be a *variable valuation*, a function that assigns to every variable an element of \mathbb{U} . TFF1 terms and formulas are evaluated according to the following equations:

$$\begin{array}{ll}
\llbracket u \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq \xi(u) & \llbracket \neg \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq \neg \llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \\
\llbracket f(\bar{\sigma}, \bar{t}) \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq f^{\mathfrak{J}}(\llbracket \bar{\sigma} \rrbracket_{\theta}^{\mathfrak{J}}, \llbracket \bar{t} \rrbracket_{\theta, \xi}^{\mathfrak{J}}) & \llbracket \varphi \wedge \psi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq \llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \wedge \llbracket \psi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \\
\llbracket p(\bar{\sigma}, \bar{t}) \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq p^{\mathfrak{J}}(\llbracket \bar{\sigma} \rrbracket_{\theta}^{\mathfrak{J}}, \llbracket \bar{t} \rrbracket_{\theta, \xi}^{\mathfrak{J}}) & \llbracket \forall u : \tau. \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq \forall a \in \llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}. \llbracket \varphi \rrbracket_{\theta, \xi[u \mapsto a]}^{\mathfrak{J}} \\
\llbracket t_1 \approx t_2 \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq (\llbracket t_1 \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \llbracket t_2 \rrbracket_{\theta, \xi}^{\mathfrak{J}}) & \llbracket \forall \alpha. \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} \triangleq \forall D \in \mathbb{D}. \llbracket \varphi \rrbracket_{\theta[\alpha \mapsto D], \xi}^{\mathfrak{J}}
\end{array}$$

We omit irrelevant subscripts and write $\llbracket \varphi \rrbracket^{\mathfrak{J}}$ to denote the evaluation of a sentence.

A sentence φ is *true* in an interpretation \mathcal{J} , written $\mathcal{J} \models \varphi$, iff $\llbracket \varphi \rrbracket^{\mathcal{J}}$ is true. The interpretation \mathcal{J} is then a *model* of φ . A sentence that has a model is *satisfiable*. A sentence that is true in every interpretation is *valid*. These notions are extended to sets and sequents of TFF1 sentences in the usual way.

4 Translation to TFF0

We describe a simple translation from TFF1 to the classical many-sorted first-order logic TFF0. The translation is included here for illustrative purposes; more suitable encoding schemes for applications are discussed in Section 7.

Our strategy for translating types is to encode them as terms and use a special binary predicate to encode type information. To avoid mixing types and terms in the encoded problem, we introduce two sorts, D and U, corresponding to the set of domains \mathbb{D} and the universe \mathbb{U} , respectively.⁴

Let Δ be a set of TFF1 sentences. We construct an equisatisfiable set of monomorphic two-sorted formulas $\mathcal{M}(\Delta)$ as follows. To every type variable α in Δ or in the type signature of a function symbol, we assign a fresh variable $\hat{\alpha}$ of sort D. To every term variable u , we assign a fresh variable \hat{u} of sort U. To every n -ary type constructor κ , we assign a function symbol $\hat{\kappa} : D^n \rightarrow D$. To every function symbol $f : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$, we assign a function symbol $\hat{f} : D^m \times U^n \rightarrow U$. To every predicate symbol $p : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow o$, we assign a predicate symbol $\hat{p} : D^m \times U^n \rightarrow o$. Finally, we introduce the special predicate symbol $\text{Ty} : U \times D \rightarrow o$.

The \mathcal{M} transformation translates TFF1 types, terms, and formulas according to the following equations:

$$\begin{aligned} \mathcal{M}(\alpha) &\triangleq \hat{\alpha} & \mathcal{M}(\kappa(\bar{\sigma})) &\triangleq \hat{\kappa}(\mathcal{M}(\bar{\sigma})) \\ \mathcal{M}(u) &\triangleq \hat{u} & \mathcal{M}(\neg \varphi) &\triangleq \neg \mathcal{M}(\varphi) \\ \mathcal{M}(f(\bar{\sigma}, \bar{\tau})) &\triangleq \hat{f}(\mathcal{M}(\bar{\sigma}), \mathcal{M}(\bar{\tau})) & \mathcal{M}(\varphi \wedge \psi) &\triangleq \mathcal{M}(\varphi) \wedge \mathcal{M}(\psi) \\ \mathcal{M}(p(\bar{\sigma}, \bar{\tau})) &\triangleq \hat{p}(\mathcal{M}(\bar{\sigma}), \mathcal{M}(\bar{\tau})) & \mathcal{M}(\forall u : \tau. \varphi) &\triangleq \forall \hat{u}. \text{Ty}(\hat{u}, \mathcal{M}(\tau)) \rightarrow \mathcal{M}(\varphi) \\ \mathcal{M}(t_1 \approx t_2) &\triangleq \mathcal{M}(t_1) \approx \mathcal{M}(t_2) & \mathcal{M}(\forall \alpha. \varphi) &\triangleq \forall \hat{\alpha}. \mathcal{M}(\varphi) \end{aligned}$$

The \mathcal{M} transformation is lifted to lists of types or terms in the evident, pointwise way.

The *inhabitation* (or *nonemptiness*) *axiom*, INH, is the formula $\forall \hat{\alpha}. \exists \hat{u}. \text{Ty}(\hat{u}, \hat{\alpha})$. For each function symbol $f : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$, the associated *typing axiom* is the formula

$$\forall \hat{\alpha}_1 \dots \hat{\alpha}_m. \forall \hat{u}_1 \dots \hat{u}_n. \text{Ty}(\hat{f}(\hat{\alpha}_1, \dots, \hat{\alpha}_m, \hat{u}_1, \dots, \hat{u}_n), \mathcal{M}(\tau))$$

We let AX_{Δ} denote the set of typing axioms associated with all function symbols occurring in Δ . Finally, we define

$$\mathcal{M}(\Delta) \triangleq \{\mathcal{M}(\varphi) \mid \varphi \in \Delta\} \cup \text{AX}_{\Delta} \cup \{\text{INH}\}$$

⁴ In fact, because the generated TFF0 problems are monotonic, the sorts can be safely erased to produce an equisatisfiable FOF problem [9].

It is easy to see that \mathcal{M} converts TFF1 types into well-formed TFF0 terms of sort D, TFF1 terms into well-formed TFF0 terms of sort U, and TFF1 formulas into well-formed TFF0 formulas.

Theorem 1 (Soundness of \mathcal{M}). *If a set of sentences Δ is satisfiable in TFF1, then $\mathcal{M}(\Delta)$ is satisfiable in TFF0.*

Proof. Let \mathfrak{M} be a model of Δ . We construct an interpretation \mathfrak{J} of $\mathcal{M}(\Delta)$ as follows. Let \mathbb{D} and \mathbb{U} stand for the set of domains and the universe in \mathfrak{M} , respectively. In \mathfrak{J} , we define the domain of sort D to be \mathbb{D} and the domain of sort U to be \mathbb{U} . Each type constructor $\hat{\kappa}$, function symbol \hat{f} , or predicate symbol \hat{p} is interpreted in \mathfrak{J} exactly as the type constructor κ , function symbol f , or predicate symbol p in \mathfrak{M} . The predicate symbol Ty is interpreted as the membership relation.

We show that for every type σ , term t , and formula φ occurring in Δ , and for every variable valuation ν in \mathfrak{J} , we have

$$\llbracket \mathcal{M}(\sigma) \rrbracket_{\nu}^{\mathfrak{J}} = \llbracket \sigma \rrbracket_{\theta}^{\mathfrak{M}} \quad \llbracket \mathcal{M}(t) \rrbracket_{\nu}^{\mathfrak{J}} = \llbracket t \rrbracket_{\theta, \xi}^{\mathfrak{M}} \quad \llbracket \mathcal{M}(\varphi) \rrbracket_{\nu}^{\mathfrak{J}} = \llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{M}}$$

where $\theta(\alpha) \triangleq \nu(\hat{\alpha})$ and $\xi(u) \triangleq \nu(\hat{u})$ for every type variable α and every variable u . We prove these equalities by induction on the structure of types, terms, and formulas. The only nontrivial case is that of a quantified formula, $\forall u : \tau. \varphi$. It is easy to see that any $a \in \mathbb{U}$ belongs to $\llbracket \tau \rrbracket_{\theta}^{\mathfrak{M}}$ iff $\llbracket \text{Ty}(\hat{u}, \mathcal{M}(\tau)) \rrbracket_{\nu'}^{\mathfrak{J}}$, where $\nu' = \nu[\hat{u} \mapsto a]$. Indeed, the latter is equivalent to $a \in \llbracket \mathcal{M}(\tau) \rrbracket_{\nu'}^{\mathfrak{J}}$. Since ν and ν' produce the same θ in \mathfrak{M} , $\llbracket \mathcal{M}(\tau) \rrbracket_{\nu'}^{\mathfrak{J}} = \llbracket \tau \rrbracket_{\theta}^{\mathfrak{M}}$. Then $\llbracket \forall \hat{u}. \text{Ty}(\hat{u}, \mathcal{M}(\tau)) \rightarrow \mathcal{M}(\varphi) \rrbracket_{\nu}^{\mathfrak{J}}$ is exactly $\llbracket \forall u : \tau. \varphi \rrbracket_{\theta, \xi}^{\mathfrak{M}}$ by induction hypothesis.

Now we must only ensure that the typing axioms AX_{Δ} and the inhabitation axiom INH hold in \mathfrak{J} . This immediately follows from the definition of interpretation in TFF1 and the first equality above. \square

Theorem 2 (Completeness of \mathcal{M}). *Given a set of sentences Δ , if $\mathcal{M}(\Delta)$ is satisfiable in TFF0, then Δ is satisfiable in TFF1.*

Proof. Let \mathfrak{M}_0 be a model of $\mathcal{M}(\Delta)$. We first construct a model \mathfrak{M} of $\mathcal{M}(\Delta)$ from \mathfrak{M}_0 by replacing every element d in the domain of D by the set $D = \{\langle a, d \rangle \mid \exists a. \text{Ty}^{\mathfrak{M}_0}(a, d)\}$ and updating the interpretations of type constructors and function and predicate symbols in \mathfrak{M}_0 accordingly. We can safely perform this substitution: Since the inhabitation axiom INH holds in \mathfrak{M}_0 , every set D is nonempty, and distinct elements are mapped to distinct sets. The predicate $\text{Ty}^{\mathfrak{M}}(a, D)$ holds iff D contains a pair $\langle a, d \rangle$ for some d .

We construct an interpretation \mathfrak{J} of Δ as follows. The set of domains \mathbb{D} is the domain of D in \mathfrak{M} . As usual, \mathbb{U} denotes the union of all domains in \mathbb{D} . Note that some elements of the domain of U may not appear in any pair in \mathbb{U} . The symbols of Δ are interpreted in \mathfrak{J} according to the equations

$$\begin{aligned} \kappa^{\mathfrak{J}}(D_1, \dots, D_m) &\triangleq \hat{\kappa}^{\mathfrak{M}}(D_1, \dots, D_m) \\ f^{\mathfrak{J}}(D_1, \dots, D_m, \langle a_1, d_1 \rangle, \dots, \langle a_n, d_n \rangle) &\triangleq \langle \hat{f}^{\mathfrak{M}}(D_1, \dots, D_m, a_1, \dots, a_n), d \rangle \\ p^{\mathfrak{J}}(D_1, \dots, D_m, \langle a_1, d_1 \rangle, \dots, \langle a_n, d_n \rangle) &\triangleq \hat{p}^{\mathfrak{M}}(D_1, \dots, D_m, a_1, \dots, a_n) \end{aligned}$$

where κ is of arity m , $p : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \mathbf{o}$, $f : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$, and d is the fixed second coordinate of pairs in the domain

$$D = \llbracket \mathcal{M}(\tau) \rrbracket_{[\hat{\alpha}_1 \mapsto D_1, \dots, \hat{\alpha}_m \mapsto D_m]}^{\mathfrak{M}} = \llbracket \tau \rrbracket_{[\alpha_1 \mapsto D_1, \dots, \alpha_m \mapsto D_m]}^{\mathfrak{J}}$$

Since the typing axioms AX_Δ hold in \mathfrak{M} , we have $\text{Ty}^{\mathfrak{M}}(f^{\mathfrak{M}}(D_1, \dots, D_n, a_1, \dots, a_m), D)$, and therefore the result of $f^{\mathfrak{J}}$ indeed belongs to D .

Given a type context γ , a type valuation θ , and a variable valuation ξ , we say that they are *admissible* for a TFF1 formula φ if the following conditions are satisfied:

- φ is well-typed in context γ ;
- for every variable v free in φ , no type variable occurring in $\gamma(v)$ is bound in φ ;
- for every variable v free in φ , we have $\xi(v) \in \llbracket \gamma(v) \rrbracket_{\theta}^{\mathfrak{J}}$.

Obviously, any triple γ, θ, ξ is admissible for a sentence. We must show that for every type σ , term t , and formula φ , and for all γ, θ, ξ admissible for φ , we have

$$\llbracket \sigma \rrbracket_{\theta}^{\mathfrak{J}} = \llbracket \mathcal{M}(\sigma) \rrbracket_v^{\mathfrak{M}} \quad \pi_1(\llbracket t \rrbracket_{\theta, \xi}^{\mathfrak{J}}) = \llbracket \mathcal{M}(t) \rrbracket_v^{\mathfrak{M}} \quad \llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \llbracket \mathcal{M}(\varphi) \rrbracket_v^{\mathfrak{M}}$$

where π_1 stands for the first projection of a pair, $v(\hat{\alpha}) \triangleq \theta(\alpha)$, and $v(\hat{u}) \triangleq \pi_1(\xi(u))$ for every type variable α and variable u .

The proof is by induction on the structure of types, terms, and formulas. There are three nontrivial cases: equality, quantification over a term variable, and quantification over a type variable.

Let φ be an equality $t_1 \approx t_2$. Let $\langle a_1, d_1 \rangle$ be $\llbracket t_1 \rrbracket_{\theta, \xi}^{\mathfrak{J}}$ and $\langle a_2, d_2 \rangle$ be $\llbracket t_2 \rrbracket_{\theta, \xi}^{\mathfrak{J}}$. By induction hypothesis, $\llbracket \mathcal{M}(t_1) \rrbracket_v^{\mathfrak{M}} = a_1$ and $\llbracket \mathcal{M}(t_2) \rrbracket_v^{\mathfrak{M}} = a_2$. We must show $d_1 = d_2$. By assumption, φ is well-typed in γ . Thus, $\gamma \vdash t_1 : \sigma$ and $\gamma \vdash t_2 : \sigma$ for some type σ . If t_1 is a variable v , then $\llbracket t_1 \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \xi(v) \in \llbracket \gamma(v) \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \llbracket \sigma \rrbracket_{\theta}^{\mathfrak{J}}$. Otherwise, t_1 is a function application $f(\sigma_1, \dots, \sigma_m, s_1, \dots, s_n)$, for $f : \forall \alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$. Let $D_i = \llbracket \sigma_i \rrbracket_{\theta}^{\mathfrak{J}}$ for every $i \in [1, m]$. Then $\sigma = \tau[\sigma_1/\alpha_1, \dots, \sigma_n/\alpha_n]$ and $\llbracket \sigma \rrbracket_{\theta}^{\mathfrak{J}} = \llbracket \tau \rrbracket_{[\alpha_1 \mapsto D_1, \dots, \alpha_n \mapsto D_n]}^{\mathfrak{J}}$. By construction of \mathfrak{J} , $\llbracket t_1 \rrbracket_{\theta, \xi}^{\mathfrak{J}} \in \llbracket \sigma \rrbracket_{\theta}^{\mathfrak{J}}$. By the same argument, $\llbracket t_2 \rrbracket_{\theta, \xi}^{\mathfrak{J}} \in \llbracket \sigma \rrbracket_{\theta}^{\mathfrak{J}}$. Hence, $d_1 = d_2$ and $\llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \llbracket \mathcal{M}(\varphi) \rrbracket_v^{\mathfrak{M}}$.

Let φ be a quantified formula $\forall u : \tau. \psi$. We must show that for every pair $\langle a, d \rangle$ in $\llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}$, we have $\text{Ty}^{\mathfrak{M}}(a, \llbracket \mathcal{M}(\tau) \rrbracket_v^{\mathfrak{M}})$, where $v' = v[\hat{u} \mapsto a]$, and vice versa, for every a in the domain of \mathbb{U} , if $\text{Ty}^{\mathfrak{M}}(a, \llbracket \mathcal{M}(\tau) \rrbracket_v^{\mathfrak{M}})$ holds, then there exists some d such that $\langle a, d \rangle \in \llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}$. Since term variables do not occur in types, $\llbracket \mathcal{M}(\tau) \rrbracket_v^{\mathfrak{M}} = \llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}$. By construction of \mathfrak{M} , $\text{Ty}^{\mathfrak{M}}(a, \llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}})$ holds iff there exists some d' such that $\langle a, d' \rangle \in \llbracket \tau \rrbracket_{\theta}^{\mathfrak{J}}$. Now, notice that the triple $\gamma[u \mapsto \tau], \theta, \xi[u \mapsto \langle a, d \rangle]$ is admissible for ψ . Indeed, ψ is well-typed in $\gamma[u \mapsto \tau]$, and for every $v \in \text{FV}(\psi)$, we have

$$\xi[u \mapsto \langle a, d \rangle](v) \in \llbracket \gamma[u \mapsto \tau](v) \rrbracket_{\theta}^{\mathfrak{J}}$$

Also, if τ contains a type variable α bound in ψ , then α is both free and bound in φ , which violates the no-clash assumption. Then, by induction hypothesis, $\llbracket \forall u : \tau. \psi \rrbracket_{\theta, \xi}^{\mathfrak{J}}$ is exactly $\llbracket \forall \hat{u}. \text{Ty}(\hat{u}, \mathcal{M}(\tau)) \rightarrow \mathcal{M}(\psi) \rrbracket_v^{\mathfrak{M}}$.

Let φ be a quantified formula $\forall \alpha. \psi$. Let D be an element of \mathbb{D} . Under the no-clash assumption, formula ψ is well-typed in γ , and for all $v \in \text{FV}(\psi) = \text{FV}(\varphi)$, we have $\llbracket \gamma(v) \rrbracket_{\theta[\alpha \mapsto D]}^{\mathfrak{J}} = \llbracket \gamma(v) \rrbracket_{\theta}^{\mathfrak{J}}$. Then $\llbracket \varphi \rrbracket_{\theta, \xi}^{\mathfrak{J}} = \llbracket \mathcal{M}(\varphi) \rrbracket_v^{\mathfrak{M}}$ by induction hypothesis. \square

5 Preprocessing of Type Quantifiers

We describe two preprocessing steps that preserve both satisfiability and unsatisfiability of TFF1 problems. They eliminate phantom type variables in type signatures and alternating \forall/\exists type quantifier prefixes in formulas, two features that are not directly supported by ML-style formalisms (which are otherwise a good match for TFF1).

5.1 Elimination of Phantom Type Variables

ML-style formalisms, as implemented in Alt-Ergo [6], Boogie [16], HOL [12], HOL Light [13], Isabelle/HOL [19], Why3 [8], and several other systems, allow type variables in type signatures, but without explicit \forall -binders. Instead of relying on explicit type arguments, these systems determine the concrete instance of a function or predicate symbol's type signature by the types of its term arguments, the type of the result, and optional type annotations inside terms. The natural translation from TFF1 to such a formalism would map $\forall\alpha\beta. \alpha \times \beta \rightarrow \circ$ (and $\forall\beta\alpha. \alpha \times \beta \rightarrow \circ$) to $\alpha \times \beta \rightarrow \circ$, simply omitting the \forall -binders. To compensate for the missing type arguments, type annotations are sometimes needed to guide the Hindley–Milner type inference.

The difficulties arise in conjunction with phantom type variables: If $\forall\alpha. \circ$ collapses to \circ , the dependency on the type is lost. To ease the adoption of TFF1 in such systems, we suggest a preprocessing step that eliminates phantom type variables. This step is lightweight, as it requires only the introduction of one term argument per phantom type. In particular, it is the identity for formulas that do not rely on phantom type variables.

Let Δ be a set of sentences. We construct an equisatisfiable set $\mathcal{E}(\Delta)$ as follows. We introduce a special unary type constructor Ph . We replace every function symbol $f : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \tau$ with a new function symbol $\hat{f} : \forall\alpha_1 \dots \alpha_m. \text{Ph}(\alpha_{i_1}) \times \dots \times \text{Ph}(\alpha_{i_r}) \times \tau_1 \times \dots \times \tau_n \rightarrow \tau$, where $\alpha_{i_1}, \dots, \alpha_{i_r}$ are the type variables from the quantifier prefix that do not occur in τ_1, \dots, τ_n or τ . Every predicate symbol $p : \forall\alpha_1 \dots \alpha_m. \tau_1 \times \dots \times \tau_n \rightarrow \circ$ is replaced with a new predicate symbol $\hat{p} : \forall\alpha_1 \dots \alpha_m. \text{Ph}(\alpha_{i_1}) \times \dots \times \text{Ph}(\alpha_{i_r}) \times \tau_1 \times \dots \times \tau_n \rightarrow \circ$, where $\alpha_{i_1}, \dots, \alpha_{i_r}$ are the type variables from the quantifier prefix that do not occur in τ_1, \dots, τ_n . Finally, we add a “witness” constant $\text{Wt} : \forall\alpha. \text{Ph}(\alpha)$. The \mathcal{E} transformation translates terms and atomic formulas as follows:

$$\begin{aligned} \mathcal{E}(u) &\triangleq u & \mathcal{E}(s \approx t) &\triangleq \mathcal{E}(s) \approx \mathcal{E}(t) \\ \mathcal{E}(f(\sigma_1, \dots, \sigma_m, \bar{t})) &\triangleq \hat{f}(\sigma_1, \dots, \sigma_m, \text{Wt}(\sigma_{i_1}), \dots, \text{Wt}(\sigma_{i_r}), \mathcal{E}(\bar{t})) \\ \mathcal{E}(p(\sigma_1, \dots, \sigma_m, \bar{t})) &\triangleq \hat{p}(\sigma_1, \dots, \sigma_m, \text{Wt}(\sigma_{i_1}), \dots, \text{Wt}(\sigma_{i_r}), \mathcal{E}(\bar{t})) \end{aligned}$$

Given a formula φ or a set of sentences Δ , $\mathcal{E}(\varphi)$ and $\mathcal{E}(\Delta)$ denote the result of applying \mathcal{E} to every atomic formula in φ and Δ , respectively.

Theorem 3. *Any Δ is equisatisfiable to $\mathcal{E}(\Delta)$.*

Proof. A model of Δ can be converted to a model of $\mathcal{E}(\Delta)$ as follows. We choose an arbitrary value e and take the singleton $\{e\}$ as the domain of $\text{Ph}(D)$ for any domain D . Accordingly, Wt evaluates to e on any argument. Interpretations of other symbols are adjusted in the obvious way. Conversely, we evaluate any function symbol f on $D_1, \dots, D_m, a_1, \dots, a_n$ in Δ exactly as \hat{f} on $D_1, \dots, D_m, e_1, \dots, e_r, a_1, \dots, a_n$ in $\mathcal{E}(\Delta)$, where each e_k is the evaluation of Wt on D_{i_k} , and similarly for predicate symbols. \square

5.2 Type Skolemization

Another feature of TFF1 that is not universally supported by systems with polymorphic types is explicit quantification over type variables. For example, while Boogie [16] and Coq [3] allow quantifiers over types, HOL and other ML-style logics provide no such syntax: They consider all type variables implicitly universally quantified at the top of a formula, which recalls the treatment of term variables in TPTP CNF clauses.

The solution is type skolemization. We define two transformations, \mathcal{S}^+ and \mathcal{S}^- , to perform type skolemization in axioms and in conjectures, respectively:

$$\begin{array}{ll}
\mathcal{S}^+(p(\bar{\sigma}, \bar{\tau})) \triangleq p(\bar{\sigma}, \bar{\tau}) & \mathcal{S}^-(p(\bar{\sigma}, \bar{\tau})) \triangleq p(\bar{\sigma}, \bar{\tau}) \\
\mathcal{S}^+(t_1 \approx t_2) \triangleq t_1 \approx t_2 & \mathcal{S}^-(t_1 \approx t_2) \triangleq t_1 \approx t_2 \\
\mathcal{S}^+(\neg \varphi) \triangleq \neg \mathcal{S}^-(\varphi) & \mathcal{S}^-(\neg \varphi) \triangleq \neg \mathcal{S}^+(\varphi) \\
\mathcal{S}^+(\varphi \wedge \psi) \triangleq \mathcal{S}^+(\varphi) \wedge \mathcal{S}^+(\psi) & \mathcal{S}^-(\varphi \wedge \psi) \triangleq \mathcal{S}^-(\varphi) \wedge \mathcal{S}^-(\psi) \\
\mathcal{S}^+(\forall u : \tau. \varphi) \triangleq \forall u : \tau. \varphi & \mathcal{S}^-(\forall u : \tau. \varphi) \triangleq \forall u : \tau. \varphi \\
\mathcal{S}^+(\forall \alpha. \varphi) \triangleq \forall \alpha. \mathcal{S}^+(\varphi) & \mathcal{S}^-(\forall \alpha. \varphi) \triangleq \mathcal{S}^-(\varphi[\kappa(\bar{\beta})/\alpha])
\end{array}$$

where κ is a fresh type constructor and $\bar{\beta}$ is the list of free type variables of $\forall \alpha. \varphi$. Since TFF1 forbids quantifiers over type variables in the scope of a quantifier over a term variable, \mathcal{S}^+ and \mathcal{S}^- simply stop at the outermost term quantifier.

To simplify the presentation, we viewed equivalence (\leftrightarrow) and inequivalence (\nleftrightarrow) as abbreviations. A slight complication arises in practice when processing type quantifiers under either connective, where they are unpolarized. The easiest solution is to expand the connective before skolemizing type quantifiers that appear under it.

Given a set of TFF1 sentences Δ , $\mathcal{S}^+(\Delta)$ and $\mathcal{S}^-(\Delta)$ denote the result of applying the corresponding transformations to every formula in Δ .

Theorem 4. *Any Δ is equisatisfiable to $\mathcal{S}^+(\Delta)$.*

Proof. It is easy to see that skolemizing the D-sorted variables in $\mathcal{M}(\Delta)$ gives exactly $\mathcal{M}(\mathcal{S}^+(\Delta))$ modulo renaming of Skolem symbols and permutation of their arguments. Theorems 1 and 2 conclude the proof. \square

6 Applications

A number of applications already support TFF1. Geoff Sutcliffe has extended the TPTP World infrastructure to process TFF1 problems and solutions. This involved in particular adapting the BNF specification of the TPTP syntaxes, from which parsers are generated. Some TPTP tools still need to be ported to TFF1; this is ongoing work.

The Why3 [8] environment, which defines its own ML-like polymorphic specification language, can parse pure TFF1. Why3 translates between TFF1 and a wide range of formats, including FOF, SMT-LIB, and Alt-Ergo's native syntax. In addition, Why3's TFF1 parser is being ported to Alt-Ergo [6], so that it can directly process TFF1.

Sledgehammer [20], a tool that bridges the interactive theorem prover Isabelle/HOL and various automatic provers, has now been extended to output pure TFF1 problems

for Alt-Ergo and Why3 (in addition to FOF, TFF0, and THF0). For the first time, Sledgehammer exploits the polymorphic potential of these tools—without having to implement their (incompatible) native file formats. Moreover, using Sledgehammer, we produced 987 problems to populate the TPTP library.⁵ By extending the tool with a TFF1 parser, we hope to transform it into a versatile translator to FOF and TFF0.

7 Related Work

Formalisms. The TPTP family of formats is well established in the automated reasoning community. As part of the MPTP project [30], Urban designed private extensions of the TPTP FOF syntax with dependent types to accommodate Mizar, as well as translations; these could form the basis of a future TFF2 format. The full THF syntax [28], which is not yet finalized or implemented, also supports dependent types.

For interactive theorem provers, polymorphism is the norm rather than the exception. HOL systems [12, 13, 19] provide ML-style (rank-1) polymorphism, while Coq [3] supports dependent types and higher-rank polymorphism. The SMT solver Alt-Ergo [6] is perhaps the only automatic prover that supports polymorphism natively.

The intermediate verification language and tool Boogie 2 [16] supports a restricted form of higher-rank polymorphism (due to its polymorphic maps), and its cousin Why3 [8] provides rank-1 polymorphism. Both provide TPTP and SMT-LIB backends.

Encodings. Early descriptions of type encodings are due to Enderton [11, §4.3], Stickel [23, p. 99], and Wick and McCune [32, §4]. TFF1 type arguments are reminiscent of System F; a FOF-based encoding that exploits them is described by Meng and Paulson [18], who also present a translation of axiomatic type classes.

Considerable progress has been made lately toward sound, complete, and efficient encodings of polymorphic logics in untyped or monomorphic logics. Leino and Rümmer [16] present a translation of higher-rank polymorphism, including explicit quantifiers over types, into the many-sorted SMT-LIB syntax, while preserving interpreted types. They also show how to exploit SMT triggers to prevent unsound variable instantiations in a translation based on type arguments. Bobot and Paskevich [7] extend earlier work by Couchot and Lescuyer [10] to encode polymorphism while preserving arbitrary monomorphic types (e.g., array types). Blanchette et al. [5], building on work by Claessen et al. [9], present a lightweight encoding of polymorphic types that exploits type monotonicity. All of these translations assume elimination of phantom type variables (Section 5.1); the last two also assume type skolemization (Section 5.2).

8 Conclusion

This paper described the TPTP TFF1 format, an extension of the monomorphic TFF0 format with rank-1 polymorphism. The new format nicely complements the existing TPTP offerings. For reasoning tools that already support polymorphism, TFF1 is a portable alternative to the existing ad hoc syntaxes. But more importantly, the format

⁵ <http://www.cs.miami.edu/~tptp/TPTP/Proposals/TFF1.html>

is a vehicle to foster native polymorphism support in automatic theorem provers. The time is ripe: After many years of untyped reasoning, we have recently witnessed the rise of interpreted arithmetic embedded in monomorphic type systems. TFF1 lifts the most obvious restrictions of such type systems.

TFF1 is part of TPTP World. The TPTP library already contains nearly a thousand TFF1 problems, and although the format is in its infancy, it is supported by several applications, including the SMT solver Alt-Ergo (via Why3). Given that many applications today require polymorphism, it is likely that other reasoning tools will gradually follow suit. The annual CADE Automated System Competition (CASC), starting with the 2013 edition, will certainly have a role to play driving adoption of the format. But regardless of progress in prover technology, equipped with a concrete syntax and suitable middleware, users can already turn their favorite automatic theorem prover into a fairly efficient polymorphic prover.

Rank-1 polymorphism is, of course, no panacea. More advanced features, such as type classes and dependent types, are not catered for (although type classes can be comfortably encoded in TFF1). These are expected to be part of a future TFF2, with the proviso that there be sufficient interest from users and implementers.

Acknowledgment. The present specification is largely the result of consensus among participants of the `polymorphic-tptp-tff` mailing list, especially François Bobot, Chad Brown, Florian Rabe, Philipp Rümmer, Stephan Schulz, Geoff Sutcliffe, and Josef Urban. We are grateful to Geoff Sutcliffe, TPTP Master of Ceremonies, for giving TFF1 his benediction and adapting the TPTP BNF and other infrastructure. He and Mark Summerfield suggested many textual improvements to this paper. We also thank Viktor Kuncak, Tobias Nipkow, and Nicholas Smallbone for their support and ideas.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard—Version 2.0. In: Gupta, A., Kroening, D. (eds.) SMT 2010 (2010)
2. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0—The core of the TPTP language for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNAI, vol. 5195, pp. 441–456. Springer (2008)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq’Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer (2004)
4. Besson, F., Fontaine, P., Théry, L.: A flexible proof format for SMT: A proposal. In: Fontaine, P., Stump, A. (eds.) PxTP 2011. pp. 15–26 (2011)
5. Blanchette, J.C., Böhme, S., Smallbone, N.: Monotonicity or how to encode polymorphic types safely and efficiently. Submitted
6. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT ’08. pp. 1–5. ICPS, ACM (2008)
7. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNAI, vol. 6989, pp. 87–102. Springer (2011)
8. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011. pp. 53–64 (2011)

9. Claessen, K., Lillieström, A., Smallbone, N.: Sort it out with monotonicity—Translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, vol. 6803, pp. 207–221. Springer (2011)
10. Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE-21. LNAI, vol. 4603, pp. 263–278. Springer (2007)
11. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
12. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL—A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
13. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M.K., Camilleri, A.J. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer (1996)
14. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LCS, vol. 4646, pp. 223–237. Springer (2007)
15. Kuncak, V.: Intermediate languages—From birth to execution. Boogie 2011 (2011)
16. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer (2010)
17. McCune, W.: OTTER 3.3 reference manual. Tech. rep., Argonne National Laboratory (2003)
18. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning* 40(1), 35–60 (2008)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
20. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL-2010 (2010)
21. Prevosto, V., Waldmann, U.: SPASS+T. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) ESCoR 2006. CEUR Workshop Proceedings, vol. 192, pp. 18–33. CEUR-WS.org (2006)
22. Schumann, J.M.: Automated Theorem Proving in Software Engineering. Springer (2001)
23. Stickel, M.E.: Schubert’s steamroller problem: Formulations and solutions. *J. Autom. Reasoning* 2(1), 89–101 (1986)
24. Stickel, M.E.: Building theorem provers. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 306–321. Springer (2009)
25. Sutcliffe, G.: The SZS ontologies for automated reasoning software. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) LPAR 2008 Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
26. Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. *J. Autom. Reasoning* 43(4), 337–362 (2009)
27. Sutcliffe, G.: The TPTP World—Infrastructure for automated reasoning. In: Clarke, E., Voronkov, A. (eds.) LPAR-16. pp. 1–12. No. 6355 in LNAI, Springer (2010)
28. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formal. Reasoning* 3(1), 1–27 (2010)
29. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 406–419. Springer (2012)
30. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning* 37(1-2), 21–43 (2006)
31. Voronkov, A.: Automated reasoning: Past story and new trends. In: Gottlob, G., Walsh, T. (eds.) IJCAI 2003. pp. 1607–1612. Morgan Kaufmann (2003)
32. Wick, C.A., McCune, W.: Automated reasoning about elementary point-set topology. *J. Autom. Reasoning* 5(2), 239–255 (1989)