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

Jasmin Christian Blanchette[1] and Andrei Paskevich[2,3]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] LRI, Université Paris-Sud, CNRS, France
[3] 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. This paper introduces the TFF1 format, an extension of TFF0 with rank-1 polymorphism. 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 [15] 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) [14]. 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 [10, 18]. A growing number of reasoners can process the recently introduced TPTP "core" typed first-order form (TFF0) [17], with monomorphic types and interpreted arithmetic [9, 13], or the corresponding higher-order form (THF0) [2]. A polymorphic version of THF0, the full THF, is in the works [16].

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, and it is difficult to encode polymorphism in a complete yet also sound and efficient manner, especially in the presence of interpreted types [3, 5, 11]. Tool authors are reduced to developing their own monomorphizers and type encodings, often using suboptimal schemes. Polymorphism arguably belongs in provers, where it can be implemented simply and efficiently, as demonstrated by Alt-Ergo [4].

This paper describes 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 translation tools. The complete specification is available online.[1] The parts that TFF1 inherits from TFF0 are described in the TFF0 specification [17].

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. 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.

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 $\mathbb{R}$ [17]. 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 result type. Type signatures can take any of the following forms:

(a) a type (predefined or user-defined);
(b) the Boolean pseudotype `$o` (the result "type" of predicate symbols);
(c) $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where $\tau_1, \ldots, \tau_n$ are types and $\tilde{\tau}$ is a type or `$o`;
(d) `!>[`α_1 `: $tType, ..., `α_n `: $tType]:` ς for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς has one of the previous three forms.

In accordance with TFF0, the parentheses in form (c) are omitted if $n = 1$. The binder `!>` in form (d) denotes universal quantification. If ς is of form (c), it must be enclosed in parentheses. All type variables must be bound by a `!>`-binder.

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.

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.

[1] `http://www21.in.tum.de/~blanchet/tff1spec.pdf`

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

```
tff(bird, type, bird: $tType).
tff(list, type, list: $tType > $tType).
tff(map, type, map: ($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, type, pi: $real).
tff(is_empty, type, is_empty : !>[A : $tType]: (list(A) > $o)).
tff(cons, type, cons : !>[A : $tType]: ((A * list(A)) > list(A))).
tff(lookup, 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` $* \cdots *$ `$i) > $i` for functions and `($i` $* \cdots *$ `$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.

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 symbol with a type signature `!>[`α_1 `: $tType, ...,` α_m `: $tType]: ((`$\tau_1 * \cdots * \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 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($_, X, nil($_)))`. 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 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, to avoid dependent types.

Example. The following problem gives the general flavor of TFF1. It declares and axiomatizes `lookup` and `update` operations on maps and conjectures that `update` is idempotent for fixed keys and values.

```
tff(map, type, map : ($tType * $tType) > $tType).
tff(lookup, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).
tff(update, 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. In this section, we use standard mathematical notation to write types, terms, and formulas. We use the symbols $\times$, $\to$, and $\forall$ to write type signatures, and write o for the Boolean pseudotype `$o`. It is convenient to treat $\approx$, $\neg$, $\wedge$, and $\forall$ as logical symbols and regard $\bot$, $\top$, $\not\approx$, $\vee$, $\to$, $\leftarrow$, $\leftrightarrow$, $\not\leftrightarrow$, and $\exists$ as abbreviations. Equality could be seen as a polymorphic predicate with the type signature $\forall\alpha.\, \alpha \times \alpha \to \mathrm{o}$, 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.

Type Checking. Let γ be a *type context*, a function that maps every variable 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 $\gamma \vdash \varphi : \mathrm{o}$ expresses that the formula φ is *well-typed* in γ. We write $f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau$ and $p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o}$ to specify type signatures of function and predicate symbols, where m and n can be 0. The typing rules of TFF1 are as follows (where ρ is a type substitution):

$$\frac{}{\gamma \vdash u : \gamma(u)}$$

$$\frac{f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau \qquad \gamma \vdash t_1 : \tau_1 \rho \quad \cdots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash f(\alpha_1 \rho, \ldots, \alpha_m \rho, t_1, \ldots, t_n) : \tau \rho}$$

$$\frac{p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o} \qquad \gamma \vdash t_1 : \tau_1 \rho \quad \cdots \quad \gamma \vdash t_n : \tau_n \rho}{\gamma \vdash p(\alpha_1 \rho, \ldots, \alpha_m \rho, t_1, \ldots, t_n) : \mathrm{o}}$$

$$\frac{\gamma \vdash s : \tau \qquad \gamma \vdash t : \tau}{\gamma \vdash s \approx t : \mathrm{o}} \qquad \frac{\gamma \vdash \varphi : \mathrm{o} \qquad \gamma \vdash \psi : \mathrm{o}}{\gamma \vdash \varphi \wedge \psi : \mathrm{o}}$$

$$\frac{\gamma \vdash \varphi : \mathrm{o}}{\gamma \vdash \neg \varphi : \mathrm{o}} \qquad \frac{\gamma[u \mapsto \tau] \vdash \varphi : \mathrm{o}}{\gamma \vdash \forall u : \tau.\, \varphi : \mathrm{o}} \qquad \frac{\gamma \vdash \varphi[\alpha'/\alpha] : \mathrm{o}}{\gamma \vdash \forall \alpha.\, \varphi : \mathrm{o}}$$

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 $\forall \alpha.\, \forall u : \alpha.\, \forall \alpha.\, \forall v : \alpha.\, u \approx v$, where the types of u and v are actually different. By assuming that no type variable can be both free and bound in the same formula, we can avoid explicit renaming of type variables, and the last typing rule's premise becomes $\gamma \vdash \varphi : \mathrm{o}$.

Semantics. An interpretation $\mathfrak{I}$ 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 $\kappa^{\mathfrak{I}} : \mathbb{D}^n \to \mathbb{D}$. Let θ be a *type valuation*, a function that maps every type variable to a domain. Types are evaluated according to the equations $[\![\alpha]\!]^{\mathfrak{I}}_{\theta} \triangleq \theta(\alpha)$ and $[\![\kappa(\tau_1, \ldots, \tau_n)]\!]^{\mathfrak{I}}_{\theta} \triangleq \kappa^{\mathfrak{I}}\big([\![\tau_1]\!]^{\mathfrak{I}}_{\theta}, \ldots, [\![\tau_n]\!]^{\mathfrak{I}}_{\theta}\big)$. Since type evaluation depends only on the values of θ on the type variables occurring in a type, we write $[\![\tau]\!]^{\mathfrak{I}}$ to denote the domain of a ground type τ.

A predicate symbol $p : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \mathrm{o}$ is interpreted as a relation $p^{\mathfrak{I}} \subseteq \mathbb{D}^m \times \mathbb{U}^n$. A function symbol $f : \forall \alpha_1 \ldots \alpha_m.\, \tau_1 \times \cdots \times \tau_n \to \tau$ is interpreted as a function $f^{\mathfrak{I}}$ on $\mathbb{D}^m \times \mathbb{U}^n$ that maps any m domains $D_1, \ldots, D_m$ and n universe elements to an element of $[\![\tau]\!]^{\mathfrak{I}}_{\theta}$, 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:

$$[\![u]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \xi(u) \qquad [\![\neg \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \neg\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi}$$

$$[\![f(\bar{\sigma}, \bar{t})]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq f^{\mathfrak{I}}\big([\![\bar{\sigma}]\!]^{\mathfrak{I}}_{\theta}, [\![\bar{t}]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\varphi \wedge \psi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \wedge [\![\psi]\!]^{\mathfrak{I}}_{\theta,\xi}$$

$$[\![p(\bar{\sigma}, \bar{t})]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq p^{\mathfrak{I}}\big([\![\bar{\sigma}]\!]^{\mathfrak{I}}_{\theta}, [\![\bar{t}]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\forall u : \tau.\, \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \forall a \in [\![\tau]\!]^{\mathfrak{I}}_{\theta}.\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta,\xi[u \mapsto a]}$$

$$[\![t_1 \approx t_2]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \big([\![t_1]\!]^{\mathfrak{I}}_{\theta,\xi} = [\![t_2]\!]^{\mathfrak{I}}_{\theta,\xi}\big) \qquad [\![\forall \alpha.\, \varphi]\!]^{\mathfrak{I}}_{\theta,\xi} \triangleq \forall D \in \mathbb{D}.\, [\![\varphi]\!]^{\mathfrak{I}}_{\theta[\alpha \mapsto D],\xi}$$

4 Applications

A number of applications already support TFF1. Geoff Sutcliffe has extended the TPTP World infrastructure to process TFF1 problems and solutions. This involved adapting the Backus–Naur form specification of the TPTP syntaxes, from which parsers are generated.[2] Some TPTP tools still need to be ported to TFF1; this is ongoing work.

The Why3 [6] 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 [5, 7]. In addition, Why3's TFF1 parser is being ported to Alt-Ergo [4].

HOL(y)Hammer [8] and Sledgehammer [12] integrate various automatic provers in the proof assistants HOL Light and Isabelle/HOL. They have been extended to output pure TFF1 problems for Alt-Ergo and Why3. Using Sledgehammer, we produced 987 problems to populate the TPTP library.[3]

5 Conclusion

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

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). Work has commenced in Saarbrücken to add polymorphism to the superposition prover SPASS [19]. Given that many applications require polymorphism, other reasoning tools are likely to follow suit. The annual CADE Automated System Competition (CASC) will surely 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; higher ranks and dependent types could be part of a future TFF2.

For SMT (satisfiability modulo theories) solvers, the SMT-LIB 2 format [1] specifies a classical many-sorted logic 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; 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.

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

[2] `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`

[3] `http://www.cs.miami.edu/~tptp/TPTP/Proposals/TFF1.html`

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, Mark Summerfield, and several anonymous reviewers suggested many textual improvements to this paper. We also thank Viktor Kuncak, Tobias Nipkow, Andrei Popescu, and Nicholas Smallbone for their support and ideas. The first author's research was supported by the Deutsche Forschungsgemeinschaft project Hardening the Hammer (grant Ni 491/14-1).

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. 491–506. Springer (2008)
3. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer (2013)
4. 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)
5. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 87–102. Springer (2011)
6. 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)
7. Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE-21. LNAI, vol. 4603, pp. 263–278. Springer (2007)
8. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. Submitted
9. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer (2007)
10. Kuncak, V.: Intermediate languages—From birth to execution. Boogie 2011 (2011)
11. 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)
12. 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)
13. 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)
14. 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)
15. Sutcliffe, G.: The TPTP World—Infrastructure for automated reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNAI, vol. 6355, pp. 1–12. Springer (2010)
16. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. J. Formal. Reasoning 3(1), 1–27 (2010)
17. 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)
18. Voronkov, A.: Automated reasoning: Past story and new trends. In: Gottlob, G., Walsh, T. (eds.) IJCAI 2003. pp. 1607–1612. Morgan Kaufmann (2003)
19. Wand, D., Weidenbach, C.: Private communication (June 2012)